O'REILLY®

3rd Edition

Introducing Python

Modern Computing in Simple Packages

Bill Lubanovic

"Bill Lubanovic has achieved a tour de force, laying down the foundations for programming and then teaching you how to deal with real-life problems through the huge Python toolbox. This book is a sure path for learning how to solve problems the Python way."

Loïc Pefferkorn, open source systems engineer

"This book ranges over an immense amount of the Python programming language and third-party packages. Delivered in a witty, conversational style, it entertains and informs."

Nathan Stocks, Agile Perception

Introducing Python

The complexities of coding can often feel like a labyrinth with no exit. Hence the popularity of Python, which is among the easiest programming languages to learn, read, and write.

Introducing Python has already proven to be an integral resource for advanced beginner and intermediate developers. Now in its third edition, with new chapters covering the discovery, use, and evaluation of real-world AI models and recent ways to remarkably improve Python's performance, Bill Lubanovic's book guides you through the intricacies and capabilities of this much-loved coding language. Easy to understand and enjoyable to read, *Introducing Python* not only teaches you core concepts but also features detailed, project-oriented walk-throughs of application areas such as the web, databases, networks, and more.

- **Understand Python data structures and operations**
- **Write correct—and readable—Python code**
- **Read the Python code of others—because developers do a lot of that**
- **Take on workhorse jobs like web and database development**
- **Try some exciting new things, like current AI models**

Bill Lubanovic has been a developer for over 40 years, specializing in Linux, the web, and Python. This is his third edition of *Introducing Python*, and he also authored *FastAPI* and coauthored *Linux System Administration* (all for O'Reilly). Bill lives with his family and cat in the Sangre de Sasquatch mountains of Minnesota.

PYTHON / PROGRAMMING

US $49.99 CAN $62.99

ISBN: 978-1-098-17440-8

O'REILLY®

Praise for *Introducing Python*, Third Edition

Bill Lubanovic has achieved a tour de force, laying down the foundations for programming and then teaching you how to deal with real-life problems through the huge Python toolbox. This book is a sure path for learning how to solve problems the Python way.

—Loïc Pefferkorn, open source systems engineer

This book ranges over an immense amount of the Python programming language and third-party packages. Delivered in a witty, conversational style, it entertains and informs.

—Nathan Stocks, Agile Perception

The most valuable thing about this book is how it presents Python's intersection with other crucial domains (AI, databases, webservers, and more) in today's rapidly changing world.

—Patrick Viafore, author of Robust Python *(O'Reilly, 2021)*

THIRD EDITION

Introducing Python

Modern Computing in Simple Packages

Bill Lubanovic

O'REILLY®

Introducing Python
by Bill Lubanovic

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

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

Acquisitions Editor: Michelle Smith
Development Editor: Corbin Collins
Production Editor: Kristen Brown
Copyeditor: Sharon Wilkey
Proofreader: Piper Content Partners
Indexer: BIM Creatives, LLC
Cover Designer: Karen Montgomery
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

November 2014: First Edition
November 2019: Second Edition
September 2025: Third Edition

Revision History for the Third Edition
2025-09-10: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098174408* for release details.

978-1-098-17440-8

[LSI]

With love to Lettie, Maeve, and Will. Oh, the things you'll see!

Table of Contents

Part III. Quests

Preface

This is the third edition of a book introducing you to one of the world's most popular programming languages: Python. You may be a beginning programmer, or have some experience and want to add Python to the languages you already know. Throughout the book, I'll sometimes contrast Python with other languages, to catch assumptions about how it works, especially with subtle differences.

Computing languages are easier to learn than human languages—they're more concise *and* precise. Python is recognized as one of the easiest computing languages to learn, read, and write. It consists of *data* (like nouns in spoken languages), as well as *instructions*, or *code* (like verbs). In alternating chapters, you'll be introduced to Python's basic code and data structures, learn how to combine them, and build up to more advanced ones. The programs that you read and write will get longer and more complex.

You'll learn the language and what to do with it. We'll begin with the core Python language and its "batteries included" standard library, and advance to finding, downloading, installing, and using some good third-party packages. My emphasis is on whatever I've found useful in 20 years of production Python development, rather than fringe topics or complex hacks.

Although this is an introduction, some advanced topics are included because I want to expose you to them. Areas like databases and the web are still covered, but technology changes fast. A Python programmer might now be expected to know something about machine learning, queues, or Unicode. You'll find details here on all of these.

Python has special features that work better than adapting styles from other languages you may know. For example, using `for` and *iterators* is a more direct way of making a loop than manually incrementing a counter variable.

When you're learning new material, it's hard to tell which terms are specific rather than colloquial, and which concepts are truly important. In other words, "Is this on

the test?" I'll highlight terms and ideas that have specific meaning or importance in Python, but not too many at once. Real Python code is included early and often.

I'll include a note such as this when the content might be confusing, or if there's a more appropriate *Pythonic* way to do it.

Python isn't perfect. I'll show you features that seem odd or that should be avoided—and offer alternatives you can use instead.

Audience

Although prior programming experience may be helpful, I want to make it possible for beginning programmers to benefit from this book. Python is an excellent first computing language, and you don't need to read and understand all of this book to get started.

Changes in the Third Edition

Although this largely follows the shape of the second edition, I've brought every page up to date:

- Dropped chapters 20–22, and appendices A, C, and E
- Added chapters on AI, data science, and performance
- Expanded coverage of development environments
- Added discussion of recent Python features and fixes
- Emphasized the use of typing hints
- Updated many examples and arcane tidbits[1]
- Expanded old Chapter 19 (*Be a Pythonista*) into a full Part II

Outline

Part I (Chapters 1–12) explores the Python fortress, the basics of the Python language. You should read these chapters in order. I work up from the simplest data and code structures, combining them on the way into more detailed and realistic programs. You can try all the code in this part on your own machine.

1 A good name for a band or pet food?

Chapter 1, "Introduction"
: Computer programs are not that different from directions that you see every day. Some little Python programs give you a glimpse of the language's looks, capabilities, and uses in the real world. You'll see how to run a Python program within its *interactive interpreter* (or *shell*) or from a text *file* saved on your computer.

Chapter 2, "Types and Variables"
: Computer languages mix data and instructions. Different *types* of data are stored and treated differently by the computer. They may allow their *values* to be changed (*mutable*) or not (*immutable*). In a Python program, data can be *literal* (numbers like `78`, text strings like `waffle`) or represented by named *variables*. Python treats variables like names or labels, which is different from many other languages and has important consequences.

Chapter 3, "Numbers"
: This chapter shows Python's simplest data types: Booleans, integers, and floating-point numbers. You'll also learn the basic math operations. The examples use Python's interactive interpreter like a calculator.

Chapter 4, "Strings"
: Learn how to create, combine, change, retrieve, and print text strings. You'll see much more in Chapter 19.

Chapter 5, "Bytes and Bytearray"
: Many data examples are binary and can be represented in Python by the bytes and bytearray data types. Binary files are discussed in Chapter 20.

Chapter 6, "If and Match"
: We'll bounce between Python's nouns (data types) and verbs (program structures) for a few chapters. Python code normally runs a line at a time, from the start to the end of a program. The `if` code structure lets you run different lines of code, depending on a data comparison.

Chapter 7, "For and While"
: We'll look at verbs again, and two ways to repeat code in a loop: `for` and `while`. You'll be introduced to a core Python concept: iterators.

Chapter 8, "Tuples and Lists"
: It's time for the first of Python's higher-level built-in data structures: lists and tuples. These are sequences of values, like LEGOs for building much more complex data structures. Step through them with iterators, and build lists quickly with comprehensions.

Chapter 9, "Dictionaries and Sets"
: Dictionaries (aka dicts) and sets let you save data by their values rather than their position. This turns out to be handy and will be among your favorite Python features.

Chapter 10, "Functions"
: Weave the data and code structures of the previous chapters to compare, choose, or repeat. Package code in functions and handle errors with exceptions.

Chapter 11, "Objects"
: The term *object* is a bit fuzzy—but important in many computer languages, including Python. If you've done object-oriented programming in other languages, Python is a bit more relaxed. This chapter explains how to use objects and classes, and when it's better to use alternatives.

Chapter 12, "Modules and Packages"
: This chapter demonstrates how to scale out to larger code structures: modules, packages, and programs. You'll see where to put code and data, how to get data in and out, handle options, tour the Python standard library, and take a glance at what lies beyond.

Part II (Chapters 13–16) goes beyond the language, into the tools and techniques you'll need to do serious Python programming. This is an expansion of the single "Be a Pythonista" chapter of the second edition.

Chapter 13, "Development Environment"
: Here, we get into virtual environments (Python version control) and package management with `pip` and other tools.

Chapter 14, "Type Hints and Documentation"
: Although they're completely optional to the Python interpreter itself, type hints really help make your code readable, and they're essential for FastAPI, Mojo, and other recent applications.

Chapter 15, "Testing"
: How do you know that your code works? Some techniques can save you a lot of time and grief.

Chapter 16, "Debugging"
: Sometimes you need to dig to find the problem.

Part III (Chapters 17–27) explores Python-inspired quests into specific application areas such as the web, databases, networks, and so on; read these chapters in any order you like.

Chapter 17, "Text Data"
: Go beyond the basic string description in Chapter 4: Unicode characters, regular expressions for text pattern matching, and more.

Chapter 18, "Binary Data"
: This area doesn't seem to get much discussion in Python books. You can do some tricky things in Python with nontextual data that you might think require a lower-level language like C.

Chapter 19, "Dates and Times"
: Dates and times can be messy to handle. This chapter shows common problems and useful solutions.

Chapter 20, "Files"
: Basic data storage uses files and directories. This chapter shows you how to create and use them.

Chapter 21, "Data in Time: Concurrency"
: This is the first hardcore system chapter. Its theme is data in time—how to use programs, processes, threads, and asyncio to do more things at a time (*concurrency*).

Chapter 22, "Data in Space: Networks"
: Send your code and data through space in networks with services, protocols, and APIs. Examples range from low-level TCP sockets, to messaging libraries and queuing systems, to cloud deployment.

Chapter 23, "Data in a Box: Persistent Storage"
: Data can be stored and retrieved with basic flat files and directories within filesystems. They gain some structure with common text formats such as CSV, JSON, and XML. As data gets larger and more complex, it needs the services of *databases*—traditional *relational* ones and newer NoSQL data stores.

Chapter 24, "The Web"
: The web gets its own chapter—clients, servers, APIs, and frameworks. You'll crawl and scrape websites and then build web applications with request parameters and templates.

Chapter 25, "Data Science"
: Python is right at home here, with a wide variety of tools and methods that are used in production systems every day.

Chapter 26, "AI"
: This new chapter is devoted to the timely subject of artificial intelligence. Python has risen to prominence largely from its heavy use in data modeling and AI development. This chapter documents various approaches to using and developing AI-based systems.

Chapter 27, "Performance"
: This is another new chapter, showing various methods of speeding up Python when it isn't frisky enough. I'll introduce a recent Python superset called Mojo, which I think could have quite an impact if its development plans go well.

The Appendix, "Practice Answers", contains answers to the end-of-chapter exercises.

Python Versions

Computer languages change over time as developers add features and fix mistakes. The examples in this book were written and tested with Python version 3.13, the most current as this book was being edited, and I'll talk about its notable additions. The "What's New in Python" page is a technical reference—a bit heavy when you're just starting with Python, but may be useful in the future if you ever need to deal with different Python versions.

About the Author

I'm a self-taught programmer, which has been mostly a good thing. I started by learning FORTRAN in 1975—with punch cards! It seems like another century now.[2] In 1977, I first encountered Unix and C, and I was hooked. When I finally ran out of money and time in graduate school (doing biomedical research on circadian rhythms), I was able to get a series of Unix jobs, such as developing graphical user interface (GUI) systems in the early '80s, before their appearance in the Apple Mac and Microsoft Windows. I worked with the early noncommercial internet until it went public in the early '90s. After the release then of Mosaic (the first cross-platform browser), I felt that the web would be the "next big thing" and pivoted to web development.

I created two startups, and developed early websites with C and two newer languages: Perl (invented in 1987 by Larry Wall) and PHP (by Rasmus Lerdorf, in 1995). I had read about Python and finally played with it for a week or two. It was much better than I expected. I was able to mock up most of a large application that four of us had written previously in C over a year—including database access and a graphical user

2 Oh, wait. It was.

display. It was time for another pivot, this time to Python, which led to the words you're reading now.

One advantage of my chance early exposure to computing is that many of the computing standards we take for granted now—relational databases, the internet, the web, GUIs, object-oriented programming—came along after I'd already been a professional programmer for a few years. I could watch each new discovery play out a bit before deciding how much faith to accord it. Even today, when you read about some cool-sounding tech (say, microservices or serverless), look for war stories:

- What are its limits?
- How do you fix it when it breaks?
- How does it perform?
- What are the security implications?

So, in this book, you may see that my opinions on some subjects (such as object inheritance, or MVC and REST designs for the web) vary a bit from the common wisdom. Decide for yourself!

Also, sometimes I use meaningless variable names like `a`, `b`, and `x`—but recommend elsewhere that you use meaningful names. I do this to keep a code example simple and short, and the variable names are clearly throwaways. I also occasionally use ellipses to elide a long line. We don't want code lines to use too much space in the print edition. Paper doesn't grow on trees, you know.[3]

And thanks for jumping in. My cat may (no promises) thank you at the end of the book.

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 variables, functions, and data types.

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

3 Oh, wait. It does.

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

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

The substantial code examples and exercises in this book are available online for you to download. This book is here to help you get your job done. In general, you may use the code in this book 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 do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Introducing Python* by Bill Lubanovic (O'Reilly). Copyright 2025 Bill Lubanovic, 978-1-098-17440-8."

If you feel your use of code examples falls outside fair use or the permission given here, 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.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
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/introducing-python-3e*.

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

My sincere thanks to the reviewers and readers who helped make this better: Corbin Collins, Patrick Viafore, Jorge Reyes, Ganesh Harke, and William Jamir.

PART I
Stronghold

In the movie *Monty Python and the Holy Grail*, King Arthur and his knights come upon a castle, where they are taunted by a French knight from atop a parapet. The taunter claims that the master of his castle is named *Guido*, although the surname is a bit garbled.

I'll now exercise my counterfeit artistic license far beyond its limits and link several points:

- In 1991, the Dutch computer scientist *Guido* van Rossum invented a computer language called Python.
- The name referred to the Monty Python comedy group, instead of an esteemed scientist or reptile.
- I assert that the castle master's surname should be *van Rossum*, because this is a book about the Python computer language.
- QED (Latin for *so there*).

The first part of this book explores the Python language as a stronghold: its components, construction, strengths, peculiarities, and weaknesses.

The second part informs you how to best use the castle: tools and techniques that complement the basic structure.

Finally, the third part consists of individual campaigns and quests that could issue from the castle, some more difficult than others, but none hopeless.

And now for something completely…

CHAPTER 1
Introduction

Only ugly languages become popular. Python is the one exception.
—Don Knuth

You're here. Good.

You may already know something about computers and Python, or you may not. If I'm going too slow for you in any section, jump ahead a page or two. You won't hurt my feelings. So, let's begin.

Mysteries

Let's begin with two mini-mysteries and their solutions. What do you think the following two lines mean?

```
(Row 1): (RS) K18,ssk,k1,turn work.
(Row 2): (WS) Sl 1 pwise,p5,p2tog,p1,turn.
```

This text snippet looks technical, like some kind of computer program. Actually, it's a *knitting pattern*—specifically, a fragment describing how to turn the heel of a sock, like the ones in Figure 1-1.

Figure 1-1. Knitted socks

This fragment makes as much sense to me as a Sudoku puzzle does to my cat, but if you're a knitter, you probably understand it perfectly.

Let's try another mysterious text, found on an index card. You'll figure out its purpose right away, although you might not know its final product:

```
    1/2 c. butter or margarine
    1/2 c. cream
    2 1/2 c. flour
    1 t. salt
    1 T. sugar
    4 c. riced potatoes (cold)

Be sure all ingredients are cold before adding flour.
Mix all ingredients.
Knead thoroughly.
Form into 20 balls. Store cold until the next step.
For each ball:
  Spread flour on cloth.
  Roll ball into a circle with a grooved rolling pin.
  Fry on griddle until brown spots appear.
  Turn over and fry other side.
```

Even if you don't cook, you probably recognize that it's a *recipe*,[1] a list of food ingredients followed by directions for preparation. But what does it make? It's *lefse*, a Norwegian delicacy that resembles a tortilla (Figure 1-2). Slather on some butter and jam or whatever you like, roll it up, and enjoy.

1 Usually found only in cookbooks and cozy mysteries.

Figure 1-2. Lefse

The knitting pattern and the recipe share some features:

- A regular *vocabulary* of words, abbreviations, and symbols. Some might be familiar, others mystifying.
- Rules about what can be said, and where—*syntax*.
- A *sequence of operations* to be performed in order.
- Sometimes, a repetition of some operations (a *loop*), such as the method for frying each piece of lefse.
- Sometimes, a reference to another sequence of operations (in computer terms, a *function*). In the recipe, you might need to refer to another recipe for ricing potatoes.
- Assumed knowledge about the *context*. The recipe assumes you that know what water is and how to boil it. The knitting pattern assumes that you can knit and purl without stabbing yourself too often.
- Some *data* to be used, created, or modified—potatoes and yarn.
- The *tools* used to work with the data—pots, mixers, ovens, knitting sticks.
- An expected *result*. In our examples, something for your feet and something for your stomach. Just don't mix them up.

Whatever you call them—idioms, jargon, little languages—you see examples of them everywhere. The lingo saves time for people who know it, while mystifying the rest of us. Try deciphering a newspaper column about bridge if you don't play the game, or a scientific paper if you're not a scientist (or even if you are, but in a different field).

Little Python Programs

I used the knitting pattern and recipe to demonstrate that programming isn't that mysterious. It's largely a matter of learning the right words and the rules.

Now, it helps greatly if there aren't too many words and rules, and if you don't need to learn too many of them at once. Our brains can hold only so much at one time.

Python is all of the following:

- A computer programming language
- Created in 1991 by the Dutch computer scientist Guido van Rossum
- Grown and guided by van Rossum and others since then
- Well designed
- Easier to read and write than most alternative languages
- A core of modern data science and artificial intelligence (AI) computing
- A valuable skill in today's computing job market
- Useful for small scripts, but also huge systems like Instagram

Let's finally see a small but real computer program (Example 1-1). What do you think this does?

Example 1-1. countdown.py

```
for countdown in 5, 4, 3, 2, 1, "hey!":
    print(countdown)
```

If you guessed that it's a Python program that prints the lines

```
5
4
3
2
1
hey!
```

then you know that Python can be easier to learn than a recipe or knitting pattern. And you can practice writing Python programs from the comfort and safety of your desk, far from the hazards of hot water and pointy sticks.

The Python program has special words and symbols (`for`, `in`, `print`, commas, colons, parentheses, and so on) that are important parts of the language's *syntax* (rules). The good news is that Python has a nicer syntax, and less of it to remember, than most computer languages. It seems more natural—almost like a recipe.

Example 1-2 is another tiny Python program; it selects one Harry Potter spell from a Python *list* and prints it.

Example 1-2. spells.py

```
spells = [
    "Riddikulus!",
    "Wingardium Leviosa!",
    "Avada Kedavra!",
    "Expecto Patronum!",
    "Nox!",
    "Lumos!",
    ]
print(spells[3])
```

The individual spells are Python *strings* (sequences of text characters, enclosed in quotes). They're separated by commas and enclosed in a Python *list* that's defined by enclosing square brackets (`[` and `]`). The word `spells` is a *variable* that gives the list a name so we can do things with it. In this case, the program would print the fourth spell:

```
Expecto Patronum!
```

Why did we use `3` if we wanted the fourth? A Python list such as `spells` is a sequence of values, accessed by their *offset* from the beginning of the list. The first value is at offset `0`, and the fourth value is at offset `3`.

People count from 1, so counting from 0 might seem weird. It helps to think in terms of offsets instead of positions. Yes, this is an example of how computer programs sometimes differ from common language usage.

Lists are common data structures in Python, and Chapter 8 shows how to use them.

The program in Example 1-3 prints a quote from one of the Three Stooges, but referenced by who said it rather than its position in a list.

Example 1-3. quotes.py

```
quotes = {
    "Moe": "A wise guy, huh?",
    "Larry": "Ow!",
```

```
    "Curly": "Nyuk nyuk!",
    }
stooge = "Curly"
print(stooge, "says:", quotes[stooge])
```

If you were to run this little program, it would print the following:

```
Curly says: Nyuk nyuk!
```

The `quotes` variable names a Python *dictionary*—a collection of unique *keys* (in this example, the name of the Stooge) and associated *values* (here, a notable saying of that Stooge). Using a dictionary, you can store and look up values by name, which is often a useful alternative to a list.

The `spells` example used square brackets (`[` and `]`) to make a Python list, and the `quotes` example uses curly braces (`{` and `}`, which are no relation to Curly), to make a Python dictionary. Also, a colon (`:`) is used to associate each key in the dictionary with its value. You can read much more about dictionaries in Chapter 9.

That wasn't too much syntax at once, I hope. In the next few chapters, you'll encounter more of these little rules, a bit at a time.

Setup

Most likely, you're working on one of the three most prominent platforms:

- Microsoft Windows
- Apple macOS
- Linux or other Unix-like system

Programming is a text-heavy process. Modern operating systems highlight *graphical user interfaces* (GUIs) for most operations, but to write programs, you'll need two tools on your computer:

- A *terminal* window for typing commands
- A *text editor* for creating and modifying files

Chapter 13 discusses fancier tools like *integrated development environments* (IDEs) that provide these features and more in one application, but in Part I we need only a text-based terminal window and an editor. These are all widely available on the three main platforms that I've mentioned.

Throughout this book, I'll include platform-specific notes as needed.

Install Python

Unfortunately, many operating systems do not come with Python, so you may need to install it. Even if your system has Python, it may be an old version. I recommend getting the most recent version of Python, from the official website, *python.org*. Click the Downloads button, then the button for your platform, and follow the directions.

Upgrade Python

In computing, it's common to number software releases as *x.y.z*, where *x* is referred to as the *major* version, *y* is a *minor* (less extensive) version within *x*, and *z* is a small bug fix or other change within *y*.

As this book was written, the most recent *major.minor* release of Python was version 3.13. The major version 3 took about 10 years to replace the previous major version 2. The minor versions come out roughly annually.[2]

In your terminal window, your *shell* program (command interpreter) shows a *prompt* that indicates you can type a command right after it. The stuff that you type is shown in **boldface**. In this book, I'm using the simple `$` as the prompt. Type the command **`python -V`**. You should see something like this:

```
$ python -V
Python 3.13
```

If your Python version is earlier than 3.13, watch in this book for notes that indicate places where earlier versions might act differently.

Run Python Programs

The program `python` can be run in two ways:

- Interactively—you type a line at a time and see what happens
- Execute a Python file

Let's try both.

2 There is not expected to be a Python 4. The 2-to-3 transition was hard enough.

The Python Interactive Interpreter

Sometimes called the *Python shell* (not the same as the operating system shell that gives you that `$` prompt), this is what you get if you type just the `python` command. (I've substituted *version info* in the following output, because the actual details can be long, and differ across machines and versions):

```
$ python
Python ... version info ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Each line that starts with `>>>` is a *prompt* from the Python interpreter for you to type a line of valid Python code.

You type Python code after that prompt, and the file executes it, line by line. This is a quick way to test snippets of Python, and I'll use it throughout this book to illustrate brief examples. You can type whatever's in **`bold text`** into the Python interpreter on your machine as you read along.

Let's try it:

```
$ python
Python ... version info ...
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello? World?")
Hello? World?
>>>
```

The `print()` function is built into Python, and it writes anything that you put between the parentheses to your terminal. In this case, we gave it a text string, which Python indicates with the enclosing quote (`"`) characters.

Python Files

You save Python code in files with a *.py* extension. And you run a Python file by typing the command `python`, followed by the Python filename.

Save that single line that you typed in the previous section into a file called *hello.py* in Example 1-4.

Example 1-4. hello.py

```
print("Hello? World?")
```

Always use a simple text editor or IDE to create your Python files. Programs like Microsoft Word may add font changes, quote substitutions, and other effects. You want plain old text.

Then, in your terminal window, tell Python to run the file:

```
$ python hello.py
Hello? World?
```

Normally, we use the interactive interpreter to test little chunks of code, and Python files for anything bigger.

Built-In Python Features

Like that `print()` function, the Python interpreter has *built-in* support for the data and code structures that you'll see in the following chapters of Part I.

The Python Standard Library

When you installed the Python interpreter, you also got Python's *standard library*: a group of Python files that are officially supported for each release. These handle many of the tasks that you'll see in Part III. You access these from your program with the `import` statement, which is fully explained in Chapter 12, and you can see it in Example 1-5 coming up.

Third-Party Python Packages

You can install Python code written by anyone and can access it the same as you do the standard library. Part III includes many examples of these packages.

A Bigger Example

I once worked for the Internet Archive, which has been saving internet content for years. Its most well-known feature is the *Wayback Machine*,[3] which has been archiving web pages for years, and is an invaluable resource to find old copies of websites, even some that are long gone. Example 1-5 accesses the Wayback Machine and illustrates many of the tasks that you can do with Python, right out of the box. You'll learn all the details eventually, as you read through this book. (The line numbers are not part of the code but were added for the explanation that follows.)

3 A reference to the Sherman and Mr. Peabody segments of the old Rocky and Bullwinkle cartoons.

Example 1-5. archive.py

```
import webbrowser
import json
from urllib.request import urlopen

print("Let's find an old website.")
site = input("Type a website URL: ")
era = input("Type a year, month, and day, like 20150613: ") + "0000"
url = f"http://archive.org/wayback/available?url={site}&timestamp={era}"
response = urlopen(url)
contents = response.read()
text = contents.decode("utf-8")
data = json.loads(text)
try:
    old_site = data["archived_snapshots"]["closest"]["url"]
    print("Found this copy:", old_site)
    print("It should appear in your browser now.")
    webbrowser.open(old_site)
except:
    print("Sorry, no luck finding", site)
```

This little Python program does a lot in a few lines. (A production version of this would include more error-checking and comments.) You don't know all these terms yet, but you will within the next few chapters. Here's what's going on in each line:

1. *Import* (make available to this program) all the code from the Python *standard library* module called `webbrowser`.
2. Import all the code from the Python standard library module called `json`. This converts text data between the JSON format (see Chapter 19 later for details) and Python data structures.
3. Import only the `urlopen()` *function* from the standard library module `urllib.request`.
4. A blank line, because we don't want to feel crowded.
5. Print some initial text to your display.
6. Print a question about a URL, read what you type, and save it in a program *variable* called `site`.
7. Print another question, this time reading a year, month, and day, and then save it in a variable called `era`. Append the hour and minute for midnight (`0000`); the Wayback Machine will look earlier and later for the closest date and time the page was last grabbed.
8. Construct a string variable called `url` to make the Wayback Machine look up its copy of the site and date that you typed. This uses the *f-string* format that can embed the values of variables.

9. Connect to the web server at that URL and request a particular *web service*.
10. Get the response data and assign it to the variable `contents`.
11. *Decode* `contents` to a text string in JSON format, and assign that string to the variable `text`.
12. Convert the `text` JSON string to `data` (a Python data structure).
13. Error-checking: `try` to run the next four lines, and if any fail, run the last line of the program (after the `except`).
14. If we got back a match for this site and date, extract its value from a three-level Python *dictionary*. Notice that this line and the next three are indented. That's how Python knows that they are part of the `try` section.
15. Print the URL that we found.
16. Print what will happen after the next line executes.
17. Display the URL we found in your web browser.
18. If anything failed in the previous four lines, Python jumps down to here.
19. If the `try` code block failed, print a message and the site that we were looking for. This is indented because it should be run only if the preceding `except` line runs.

You don't need to understand all of this now! This example is meant to show how much you can do in Python in relatively few lines, which are more readable (and safer!) than a summoning spell from an ancient manuscript.

Most computing languages use character delimiters to indicate the start and end of multiline code blocks. Some use literal words like `start` and `end`, and the "curly brace" languages (C, C++, Java, JavaScript, and many others) use `{` and `}`. Python uses consistent indentation instead. It makes the code a bit less busy and easier to read.

When I ran this in a terminal window, I typed a website name and a date, and got this text output:

```
$ python archive.py
Let's find an old website.
Type a website URL: xkcd.com
Type a year, month, and day, like 20150613: 20240728
Found this copy:  http://web.archive.org/web/20240727204346/https://xkcd.com/
It should appear in your browser now.
```

And Figure 1-3 shows what appeared in my browser.

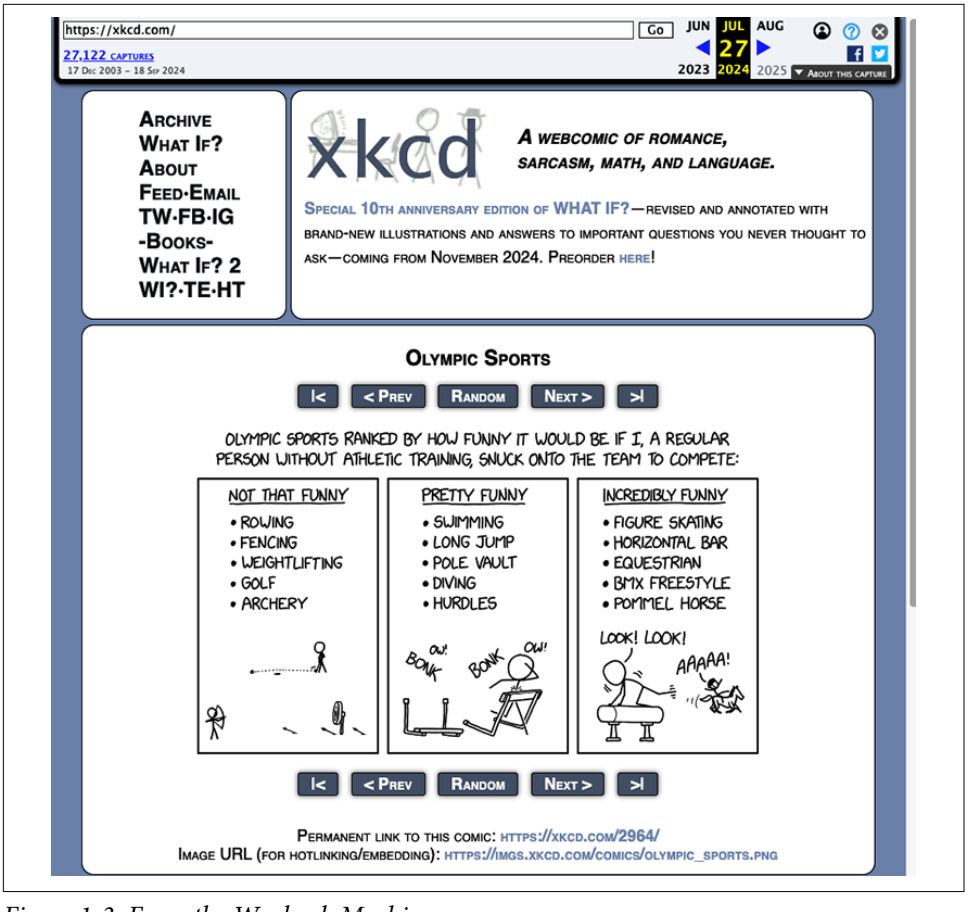

Figure 1-3. From the Wayback Machine

For various reasons, the saved archive pages are sometimes missing content, like images, that had been on the original site. At the top of this archived page is an added bar that lets you access any past archive of that site. The Archive is a great tool to learn what people actually said and did at particular times.

Review/Preview

This chapter listed the minimal tools that you'll need to write Python programs, with code examples showing some basic Python data types like lists and strings. The chapter also included a larger sample program and its output.

The next chapter gets into more of those data *types*, as well as *variables*—core concepts in any programming language.

CHAPTER 2

Types and Variables

A good name is rather to be chosen than great riches.
—Proverbs 22:1

Computer programs consist of commands (often shorthanded as *code*) and data. This chapter introduces the basics of how computers handle data, and how Python is a bit different from many programming languages.

A Computer

What's in a typical computer?

A central processing unit (CPU), or chip
: The brains of the operation. Most computers now have multiple *cores* per CPU, and we'll say more about this in Chapters 23 and 27.

Random access memory (RAM)
: The fast but limited storage area for data and programs.

Cache memory
: Like RAM, but smaller and faster.

A disk
: Stores much more data than RAM does but is thousands of times slower.

Local input devices
: Keyboards, touchscreens, mice, cameras, and microphones.

Local output devices
: Displays and printers.

A network
: Communicates with other computers, either through cables or the air.

The CPU, RAM, and cache are *volatile*: they work only while electricity is flowing through them. The disk is *nonvolatile*, retaining its data even when the whole computer is turned off. A continual struggle occurs between RAM/cache (fast but limited) and disk (slow but roomier) to balance a computer's performance.

Computer programs and data are stored on the disk and transferred to RAM when the CPU needs to access them. The CPU *fetches* (reads) from RAM and *stores* (writes) back.

In this book, I'll liken RAM to a series of bookshelves, where each rack is the same height and width, and each slot on a particular shelf is uniquely numbered. Each location is independently addressable, so you can put data on a shelf and find it later. Each slot is a *byte* wide. So I'd better explain what a byte is and get back to this bookshelf metaphor afterward.

In Chapter 27, a more complete computer architecture is discussed.

Bits and Bytes

At the very bottom, all data in a computer consists of bits. A *bit* is a tiny unit of storage that can represent one of two states. Way down in the electronics of the computer's storage (RAM or disk), bits are implemented by microscopic components and different voltages.

These two states can be interpreted in different ways, such as these:

- Set or unset
- On or off
- A number (1 or 0)
- A *Boolean* value (true or false)

Figures 2-1 and 2-2 depict the two bit possibilities.

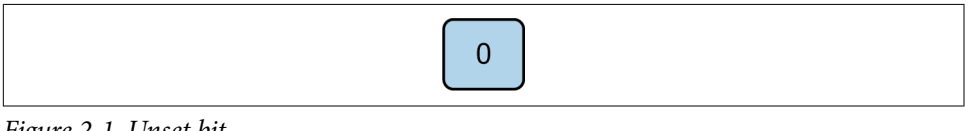

Figure 2-1. Unset bit

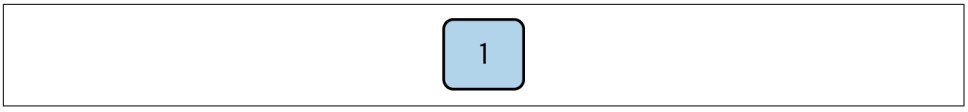

Figure 2-2. Set bit

It's hard to express more complex ideas with just two values, so, going up one level, computers bundle eight bits together into a *byte*. Because each bit has two possible values, and we're now treating eight bits as a unit, there are 2^8 (256) possible combinations of bit values in a byte.

Individual bits or bytes have no actual meaning. Each one has a combination of states, and we somehow need to keep track of them and treat them in a certain way. The bits in a byte have a particular *order*, from *least significant* (bottom) to *most significant* (top). Figure 2-3 shows a byte with the integer value 0, Figure 2-4 represents the integer 1, and Figure 2-5 has all bits set, which could represent the decimal integer 255.

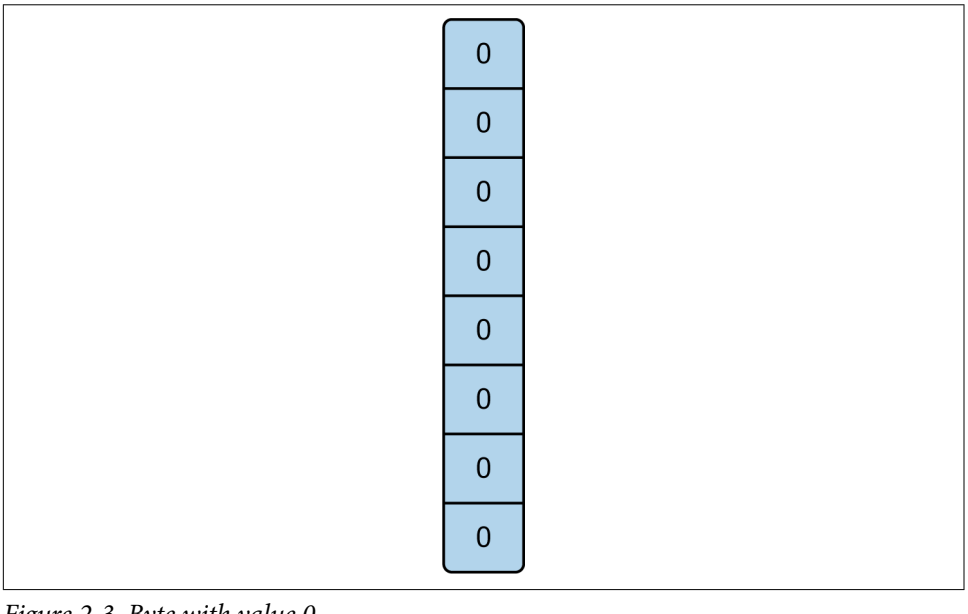

Figure 2-3. Byte with value 0

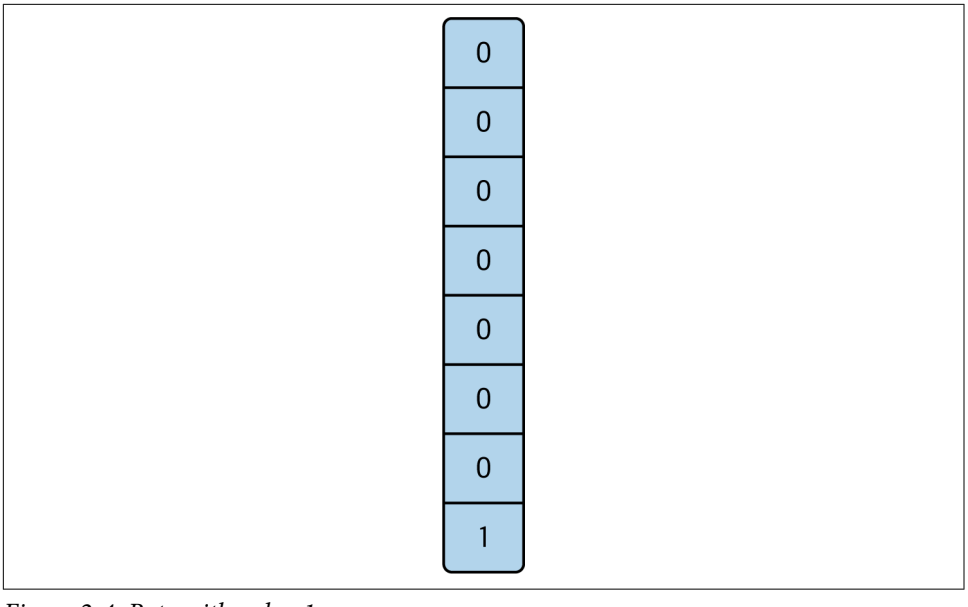

Figure 2-4. Byte with value 1

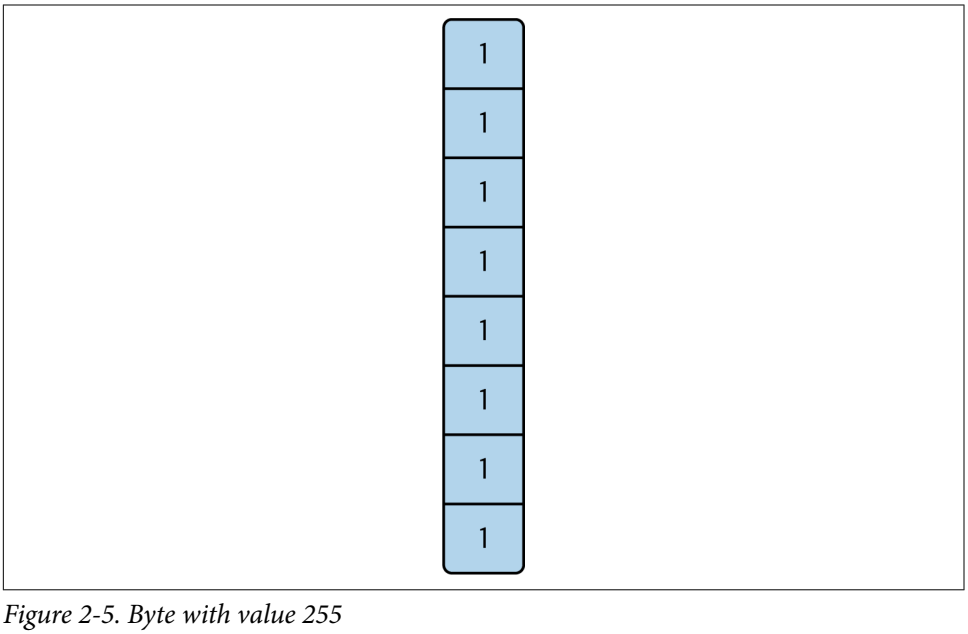

Figure 2-5. Byte with value 255

Because a byte can have up to 256 distinct combinations of bits, we can use base 2, or *binary arithmetic*, to treat a byte as a tiny number. If all its bits are 0 (off), as in Figure 2-3, that could represent the integer 0. If all bits are 1 (on), as in Figure 2-5, that could represent the decimal integer 255. The least significant bit represents how many 1s, the next bit up represents how many 2s, and the top, most significant bit represents how many 128s. (This is like our familiar base-10 (*decimal*) system, where we have digits from 0 to 9, but no single digit for 10. Instead, we have a 1 in the ten's *place*, and up to the hundred's place, and so on.) These are called *positional systems*. Setting all the bits means (from the top down): 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255.

These examples use *unsigned* (always positive) numbers, but we could treat a byte as a *signed* number (positive, negative, or zero) by using one of the bits to indicate the sign. But then we have one less bit to represent the number's *magnitude*; we can't get to positive 255 in a single byte anymore.

Finally, we could use a byte to represent a text *character*, such as a letter, digit, or punctuation symbol. English uses 26 letters, so 26 lowercase letters, 26 uppercase letters, and the digits 0 through 9 would need only 62 distinct values to fit in a byte. The rest of the 255 possible bit combinations could be used for something else.

The American Standard Code for Information Interchange (ASCII) standard was defined many decades ago to fill in these slots. It also included common punctuation symbols and some nonprinting values. It handled everything you'd see on a typewriter or computer keyboard in a single byte—but only if it was an American keyboard, because that's what the *A* in *ASCII* stands for. In Chapter 17, you'll see how *Unicode* allows all the world's languages (and symbols and even emojis) to be expressed in computers. Unicode has millions of unique symbols; we can't stuff them all into a single byte anymore.

Multibyte Types

Now, let's go back to the bookshelf metaphor that I used earlier. If you have some data that fits inside a byte, you could store it at a specific location on that shelf (location in the computer's memory).

Just as bits can be combined into bytes to represent more possible values, bytes can be combined into adjacent multibyte structures. Two bytes provide $2^8 * 2^8 = 2^{16} = 65{,}536$ unique bit combinations. Four bytes (32 bits) allows 4,294,967,296 combinations. So we can use more than one adjoining byte slot on our memory bookshelf. In Figure 2-6, four bytes in a row make a single four-byte number, with the value 1.

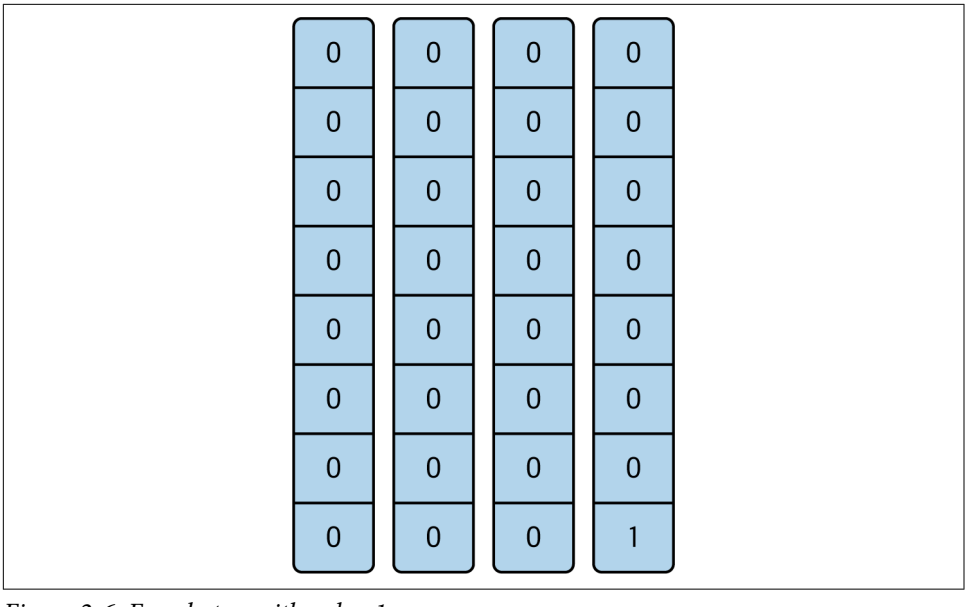

Figure 2-6. Four bytes, with value 1

The *most significant byte* is the leftmost one, so setting only the lowest bit in the least significant (rightmost) byte represents the number 1.

What would a decimal million (1,000,000) look like, represented in bits across four bytes?[1] See Figure 2-7.

In a library, a narrow book might be a novella or poetry collection, and a wider one might be a full novel or nonfiction book like a dictionary. In the computer, we might need only a narrow slot (a byte) for one piece of data, but a wider slot for something else. Unlike the library, where books have spines with titles and other identifying information, computer memory locations are anonymous. It's the job of the computer program to associate a *type* with memory locations.

1 A quick peek at two Python ways to convert that decimal 1,000,000 to binary for this image: `bin(1000000)` or `f"{1000000:b}"`.

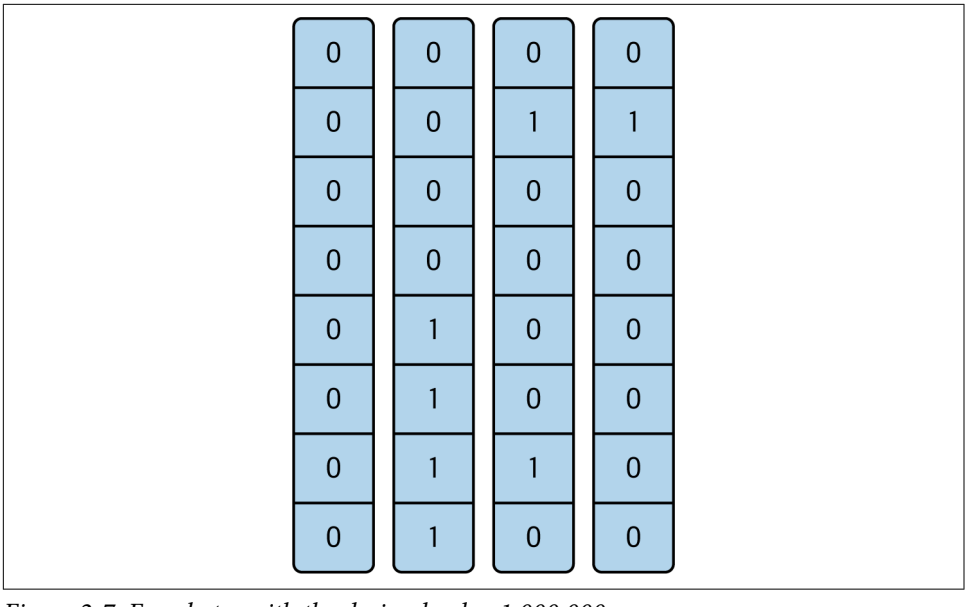

Figure 2-7. Four bytes with the decimal value 1,000,000

Variables

This giant bookshelf of bytes in computer memory contains everything that the computer's brain (its CPU) needs in order to do computery things. But how do you keep track of everything?

A Python program has code (the instructions) and data (the values and their types). Like most computing languages, Python has the concept of a *variable*: a unique name inside the code that refers to a specific data value. It keeps track of exactly where the data is stored on these long memory bookshelves.

Python handles variables differently from many other computer languages, so let's explain, with text and pictures.

Assign a Value to a Variable

You *assign* a value to a variable when you're creating it. In Python, like many other languages, you assign a value by specifying the following:

```
variable = value
```

For example:

```
x = 5
```

assigns the integer value `5` to the variable `x`.

This = is not expressing *equality*, like a formula in algebra. Maybe computer languages could have used a different symbol for assignment than = (and some do), but assignment is so common in computing that it's established itself. You quickly get used to it.

Try assigning the integer value `5` to the variable `x` in the Python shell:

```
$ python
Python 3.13.0 [version info]
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 5
>>> x
5
```

When you type just a variable on a line, the shell prints its value for you on the next line. This is a feature of the shell, and it won't happen when you execute a Python file.

Remember those memory bookshelves? One way to visualize this is that somewhere in memory, one or more bytes are used to represent the integer value `5`. Figure 2-8 uses four contiguous bytes.

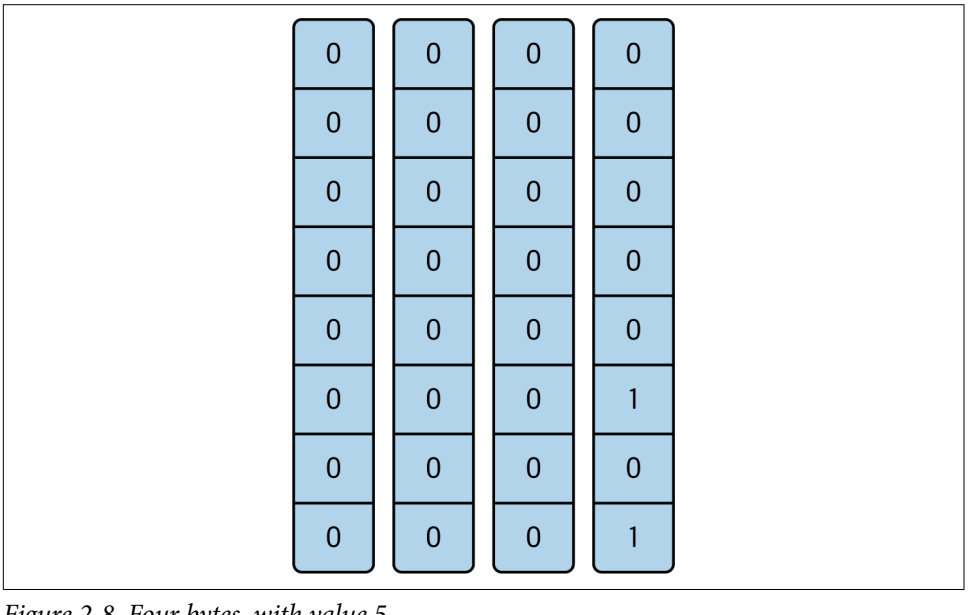

Figure 2-8. Four bytes, with value 5

OK, what about the variable `x`? That's somewhere else in memory, along with all the code and other variables for this program.

Now, here's an important distinction: in *many* programming languages (but *not* Python), a variable points to the value directly in memory. The variable in those

languages can be thought of as a box that can contain values of the same type. The variable is essentially an *address* or a *pointer*, and any valid value can be assigned to its memory destination slot and replace the previous value. Computer languages that work this way are called *statically typed*. They remember where `x` points, and that its value in memory needs to be a valid four-byte integer.

But, as I said, Python is different. Instead of tracking the type with the variable in the code, the *value* in memory is bundled with the type and extra bookkeeping info. This means that the variable is just a *name*, like a sticky note that you attach to something. Besides a *name*, a Python variable might also be called a *reference* or *label*. Python variables are *dynamically typed*.

Figure 2-9 is a simplified idea of what a Python *object* looks like in memory.

Many Python discussions mention that every data type in Python is implemented as an *object*. The problem is that *object* can have multiple meanings. You'll see how to create new data types with classes and objects in Chapter 11. When I use the term here, I mean a minimal data structure that describes the core features of this particular data, including the raw value that the CPU uses.

I'll drop the little bit boxes now and just say what's stored at each location (Figure 2-9). The actual number of bytes that each component takes up is an implementation detail that we don't need to care about.

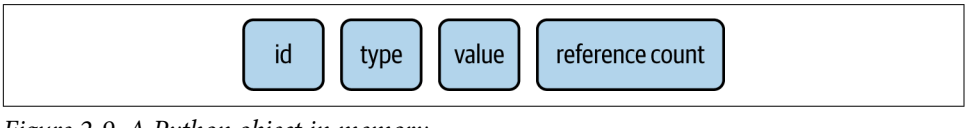

Figure 2-9. A Python object in memory

Here's what these parts represent:

- The `id` is unique for each object in memory. Although the Python interpreter may use just the object's location in memory, don't assume this. The important point is that the `id` is unique. The `id()` function returns the unique ID:

```
>>> x = 5
>>> id(x)
4488951848
```

- The `type` box indicates the Python type of this data location. Besides the built-in types that I list in Table 2-1 later in this chapter, you can define your own custom types in Python. You'll see how in Chapter 11.

- The actual value bits (in this example, those for the integer `5`) might be stored here in the `value` location or elsewhere in memory. In coming chapters, you'll see data structures that are bigger than single integers and need room to roam.
- What about that `reference count`? It's how Python tracks the number of variables pointing to the same object. If this count reaches zero, no variables are referring to this particular piece of data, and the program has no way of reaching it anymore. Python can then clear this area of memory and use it for other values; this is called *garbage collection*.

OK. What all this means is that, in Python, if you type `x = 5`, Python will create that multipart data structure (*object*) in memory with type `integer` and value `5`. Python will return this structure's memory address and associate it with the variable `x`. Then the reference count will be incremented from `0` to `1`, because its new friend, variable `x`, now knows about it.

After this in the program, any reference to the variable `x` will follow the big metaphorical finger that it points to memory and get that lovely integer `5`.

If you like visual explanations, the YouTube clip "How Variables Works in Python | Explained with Animations" by Sreekanth about variables in Python and other languages may help.

Change the Value of a Variable

The value itself can be changed by assigning a new value to the same variable. Say we make this change:

```
x = 6
```

As I mentioned earlier, in most languages (*not* Python), a variable is a pointer; that old bit pattern representing the value `5` gets wiped, and Figure 2-10 shows that it's overwritten by the bits that represent 6.

Python does this differently: it first creates a new object in memory, with type integer and the value 6. Then it associates that existing variable `x` with this new object, and accordingly increments the reference count of this new object to one.

Because `x` now refers to a new object, its `id` changes:

```
>>> x = 5
>>> id(x)
4488951848
>>> x = 6
>>> id(x)
4488951880
```

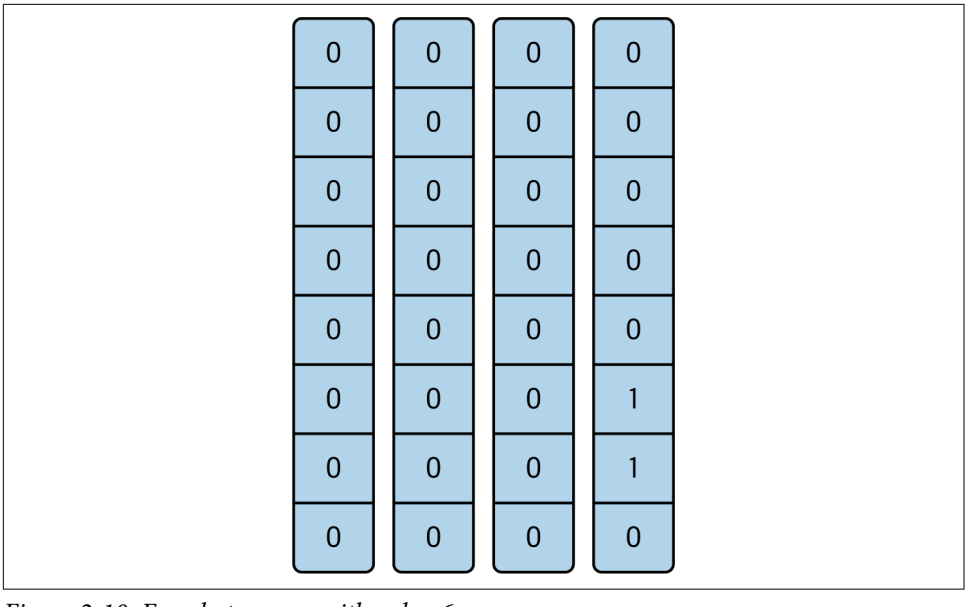

Figure 2-10. Four bytes, now with value 6

Because the variable `x` no longer refers to the previous value (5), the reference count of the old object (the one with integer value 5) is decremented to zero.

An object with a zero reference count is no longer addressable by the code, so Python eventually *garbage collects* it (frees the memory it uses).

A *very* technical exception to this arises for a value like 5. The Python interpreter has a few performance tweaks: it precreates objects for the integers –5 to 255 and keeps them around, because small numbers are so common in code. Any other objects do go on the garbage truck. Au revoir!

Delete a Variable

Another way to get rid of a value is to delete its variable: `del x`.

This also decrements its value object's reference count and causes its garbage collection. You normally don't need to do this in Python; the interpreter keeps track of the *scope* (code extent lifetime) of each variable, zeroing reference counts and freeing up the memory that they used.

Memory management is difficult and notoriously error-prone in many other languages. One of the big advantages of Python is that you don't need to manually manage memory (say this fast three times). There are no C-style `malloc()` or `free()` calls.

I'm presenting all this detail for a reason: to pre-explain some Python behavior that might be confusing if you're used to other languages, such as assigning a *mutable* (changeable) piece of data to multiple variables. You'll see this in Chapter 8 when we talk about lists.

Name Variables

Python variable names have rules:

- They can contain these characters:[2]
  - — Lowercase letters (`a` through `z`)
  - — Uppercase letters (`A` through `Z`)
  - — Digits (`0` through `9`)
  - — Underscore (`_`)
- They are *case-sensitive*: `thing`, `Thing`, and `THING` are different names.
- They must begin with a letter or an underscore, not a digit.
- They cannot be one of Python's *reserved words* (also known as *keywords*).

The reserved words are listed here:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Within Python, you can find the reserved words with

```
>>> help("keywords")
```

or

```
>>> import keyword
>>> keyword.kwlist
```

These are valid names:

- `a`
- `a1`
- `a_b_c___95`
- `_abc`
- `_1a`

2 Technically, they can also be some Unicode characters, but this can get complicated.

These names, however, are not valid:

- `1`
- `1a`
- `1_`
- `name!`
- `another-name`

Follow Naming Conventions

Python also has *conventions* when naming variables. They aren't rules, like those I just mentioned, but they'll help you be consistent with other Python code out there:

- Python variables should use *snake case*: lowercase letters (and possibly digits) separated by the underscore (_) character. Examples: `x_squared`, `num_ghosts`.
- Other languages (like Java and JavaScript) prefer *camel case* (an initial lowercase letter and uppercase humps inside): `xSquared`, `numGhosts`.
- Other languages capitalize the first letter too: `XSquared`, `NumGhosts`. Python also likes this convention when you're defining *object classes*, which are coming in Chapter 11. Sometimes this is called *Pascal case* or *upper camel case*.
- Although Python doesn't have true *constants* (variables whose value can't change), it recommends snake case with all caps to help everyone remember that this variable should *not* be modified after its initial assignment: `MAX_ITEMS`, `SECRET_CODE`.

Python really seems to like that humble underscore character:

- A name that starts with a single underscore (_) is treated as sort of private by the `import` statement (see Chapter 12).
- A name that starts with two underscores (__) is treated specially when creating object classes (see Chapter 11).
- Names that start *and* end with double underscores are used for so-called *magic*, or *dunder*, methods in object classes (also in Chapter 11).

Python Types

You've seen that bits, bytes, and multibyte combinations have no inherent meaning in computer memory or storage. Something, somewhere, assigns that meaning and remembers it. That's where computer language *types* come in.

Each computer architecture (a specific design from a computer company) has the ability to handle particular bit combinations and treat them as distinct types. These types include numbers of various sizes, text characters, and so on. Python defines its

own object types to match these common hardware types, as well as some software-only types like complex numbers. Table 2-1 lists these built-in types, with table columns that contain the following:

- The Name column contains equivalent English names.
- The Type column indicates the actual Python wording for this type.
- The Mutable column indicates whether the *value* of this type (not the type itself!) can be changed.
- The Examples column shows some Python syntax to express values of this type.
- The Chapter column is where you'll really get into the details.

Table 2-1. Python's basic data types

Name	Type	Mutable?	Examples	Chapter
Boolean	`bool`	No	`True`, `False`	Chapter 3
Integer	`int`	No	`47`, `25000`, `25_000`	Chapter 3
Floating point	`float`	No	`3.14`, `2.7e5`	Chapter 3
Complex	`complex`	No	`3j`, `5 + 9j`	Chapter 3
Text string	`str`	No	`'alas'`, `"alack"`, `'''a verse attack'''`	Chapter 4
List	`list`	Yes	`['Winken', 'Blinken', 'Nod']`	Chapter 8
Tuple	`tuple`	No	`(2, 4, 8)`	Chapter 8
Bytes	`bytes`	No	`b'ab\xff'`	Chapter 5
Bytearray	`bytearray`	Yes	`bytearray(...)`	Chapter 5
Set	`set`	Yes	`set([3, 5, 7])`	Chapter 9
Frozen set	`frozenset`	No	`frozenset(['Elsa', 'Otto'])`	Chapter 9
Dictionary	`dict`	Yes	`{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}`	Chapter 9

In the next chapter, we'll get into number types: bool, int, float, and complex.

Specify Values

By now, you've seen that a value can be specified as a *literal* (a direct value, like `5`) or a *variable* (like `x`, which was assigned the value `5`). Python has rules for specifying literal values, and they depend on the underlying types. As you've seen, an integer is specified as a sequence of digits, but a *float* (floating-point value, which you'll see in the next chapter) can consist of a string of digits and a decimal point.

In the next few chapters, I'll show how to specify literal values for Python's standard types, assign them to variables, and perform various *operations* (like the addition of numbers).

Objects as Plastic Boxes in Memory

You can think of an *object* as a clear plastic box on that memory shelf. Some content has a fixed size (ID, type, reference count), and other content varies in size (the value bits themselves). All Python *data* is like this. You could have an `int` object sitting next to a `list` object, and a user-defined object next to that.

Python *code* lives somewhere else in memory (on another shelf). A *variable* in Python code can be pictured as a sticky note attached to its data object, or a tag on a thread taped to it. Python keeps track of all this code and data so we don't have to.

Review/Preview

This chapter focused on variables (names used in programs) and objects. Python is different from many programming languages, because the variable is just a name, and the object it refers to includes other information: a unique ID, type, value, and reference count of the number of variables that refer to it.

Coming next: numbers! Yay! The simplest objects of all.

Practice

2.1 Install Python if you don't have it on your computer already. Use the most recent version if you can.

2.2 Start the interactive interpreter and type **`print(42)`**. The interpreter should echo `42` on the next line.

2.3 If you're still in the interactive interpreter, type **`43`**. The interpreter should print `43` on the next line. This is a feature of the interactive interpreter only and won't print anything if you're executing a Python file.

2.4 Try assigning the literal value `7` to variables with different names. Try some illegal names to see the error messages that Python tosses back to you.

CHAPTER 3
Numbers

> *That action is best which procures the greatest happiness for the greatest numbers.*
>
> —Francis Hutcheson

In this chapter, we begin by looking at Python's simplest built-in data types:

- *Booleans* (which have the value `True` or `False`)
- *Integers* (nonfractional numbers such as `42`, `0`, `-90`, and `100000000`)
- *Floats* (numbers with decimal points such as `3.14159`, or sometimes exponents like `1.0e8`, which means *one times ten to the eighth power*, or `100000000.0`)

In a way, these basic data types are like atoms. We use them individually in this chapter, and in later chapters you'll see how to combine them into larger "molecules" like lists and dictionaries.

Each type has specific rules for its usage and is handled differently by the computer. I also show how to use literal values like `97` and `3.1416`, and the variables that I mentioned in Chapter 2.

The code examples in this chapter are all valid Python, but they're snippets. We'll be using the Python interactive interpreter, typing these snippets and immediately seeing the results. Try running them yourself with the version of Python on your computer. You'll recognize these examples by the `>>>` prompt.

Booleans

In Python, the only values for the Boolean data type are `True` and `False`. Essentially, this is a *bit*. Sometimes you'll use Booleans directly; other times you'll evaluate the "truthiness" of other types from their values.

The special Python function `bool()` can convert any Python data type to a Boolean. Functions get their own focus in Chapter 10, but for now you just need to know that a function has the following:

- A name.
- Zero or more comma-separated input *arguments*, surrounded by parentheses. Even if the function has no arguments, you still need the parentheses (`()`).
- A *return value*.

When I refer to a function in the text, I'll include the parentheses after it to help you recognize it.

The `bool()` function takes any value as its argument and returns the Boolean equivalent.

Nonzero numbers are considered `True`:

```
>>> bool(True)
True
>>> bool(1)
True
>>> bool(45)
True
>>> bool(-45)
True
```

And zero-valued ones are considered `False`:

```
>>> bool(False)
False
>>> bool(0)
False
>>> bool(0.0)
False
```

You'll see the usefulness of Booleans in Chapters 6 and 7. In later chapters, you'll see how lists, dictionaries, and other types can be considered `True` or `False`.

Integers

Integers are whole numbers—no fractions, no decimal points, nothing fancy. Well, aside from a possible initial positive (`+`) or negative (`-`) sign. And bases, if you want to express numbers in other ways than the usual decimal (base 10); I'll say more about that in "Bases" on page 39.

Literal Integers

Any sequence of digits in Python represents a *literal integer*:

```
>>> 5
5
```

A plain zero (`0`) is valid:

```
>>> 0
0
```

But you can't have an initial `0` followed by a digit from `1` to `9`:

```
>>> 05
  File "<python-input-0>", line 1
    05
    ^
SyntaxError: leading zeros in decimal integer literals are not permitted;
use an 0o prefix for octal integers
>>>
```

`SyntaxError` is a specific kind of Python exception. An *exception* warns that you typed something that breaks Python's rules. I explain what this means in "Bases" on page 39. You'll see many more examples of exceptions in this book because they're Python's main error-handling mechanism.

You can start an integer with `0b`, `0o`, or `0x`.

A sequence of digits specifies a positive integer. If you put a plus sign (`+`) before the digits, the number stays the same:

```
>>> 123
123
>>> +123
123
```

To specify a negative integer, insert a hyphen (`-`) before the digits:

```
>>> -123
-123
```

The sign doesn't need to be right next to the digits:

```
>>> + 123
123
>>> - 123
-123
```

Sorry, you can't have any commas when typing in an integer:

```
>>> 1,000,000
(1, 0, 0)
```

Instead of a million, you'd get a *tuple* (a sequence of values, separated by a comma; see Chapter 8). Python uses the comma (`,`) to create tuples. But you *can* use the underscore character (_) as a digit separator (in Python 3.6 and newer):

```
>>> million = 1_000_000
>>> million
1000000
```

The underscore can't be the first or last character; anywhere after the first digit, it's ignored:

```
>>> 1_2_3
123
```

Integer Operations

For the next few pages, I show examples of Python acting as a simple calculator. You can do standard arithmetic with Python by using the math *operators* in Table 3-1.

Table 3-1. Integer operations

Operator	Description	Example	Result
+	Addition	`5 + 8`	`13`
-	Subtraction	`90 - 10`	`80`
*	Multiplication	`4 * 7`	`28`
/	Floating-point division	`7 / 2`	`3.5`
//	Integer (truncating) division	`7 // 2`	`3`
%	Modulus (remainder)	`7 % 3`	`1`
**	Exponentiation	`3 ** 4`	`81`

Addition and subtraction work as you'd expect:

```
>>> 5 + 9
14
>>> 100 - 7
93
>>> 4 - 10
-6
```

You can include as many numbers and operators as you'd like:

```
>>> 5 + 9 + 3
17
>>> 4 + 3 - 2 - 1 + 6
10
```

Note that you're not required to have a space between each number and operator:

```
>>> 5+9   +      3
17
```

Using a space just looks better stylewise and is easier to read.

Multiplication is also straightforward:

```
>>> 6 * 7
42
>>> 7 * 6
42
>>> 6 * 7 * 2 * 3
252
```

Division is a little more interesting because it comes in two flavors:

- `/` carries out *floating-point* (decimal) division
- `//` performs *integer* (truncating) division

Even if you're dividing an integer by an integer, using a `/` will give you a floating-point result (*floats* are coming later in this chapter):

```
>>> 9 / 5
1.8
```

Truncating integer division returns an integer answer, throwing away any remainder:

```
>>> 9 // 5
1
```

Instead of tearing a hole in the space-time continuum (the cosmos hates when that happens), dividing by zero with either kind of division causes a Python exception:

```
>>> 5 / 0
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    5 / 0
    ~~^~~
ZeroDivisionError: division by zero
>>> 7 // 0
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    7 // 0
    ~~^^~~
ZeroDivisionError: integer division or modulo by zero
```

Integers and Variables

All the preceding examples use literal integers. You can mix literal integers and variables that have been assigned integer values:

```
>>> a = 95
>>> a
95
>>> a - 3
92
```

You'll remember from Chapter 2 that `a` is a name that points to an integer object. When I said `a - 3`, I didn't assign the result back to `a`, so the value of `a` did not change:

```
>>> a
95
```

If you wanted to change `a`, you would do this:

```
>>> a = a - 3
>>> a
92
```

Again, this would not be a legal math equation, but it's how you reassign a value to a variable in Python. In Python, the expression on the right side of the equals sign (`=`) is calculated first, and then assigned to the variable on the left side.

If it helps, think of it this way:

1. Subtract `3` from `a`.
2. Assign the result of that subtraction to a temporary variable.
3. Assign the value of the temporary variable to `a`:

```
>>> a = 95
>>> temp = a - 3
>>> a = temp
```

So, when you say

```
>>> a = a - 3
```

Python is calculating the subtraction on the righthand side, remembering the result, and then assigning it to `a` on the left side of the `+` sign. This approach is faster and neater than using a temporary variable.

You can combine the arithmetic operators with assignment by putting the operator before the `=`. Here, `a -= 3` is like saying `a = a - 3`:

```
>>> a = 95
>>> a -= 3
```

```
>>> a
92
```

This is like `a = a + 8`:

```
>>> a = 92
>>> a += 8
>>> a
100
```

And this is like `a = a * 2`:

```
>>> a = 100
>>> a *= 2
>>> a
200
```

Again, plain integer division (using a single `/`) always produces a float, even if there's no remainder:

```
>>> a = 200
>>> a /= 4
>>> a
50.0
>>> a = 200
>>> a /= 3
>>> a
66.66666666666667
```

Truncating integer division (which uses `//`) always tosses the remainder and produces an integer:

```
>>> a = 13
>>> a //= 4
>>> a
3
```

The `%` character has multiple uses in Python. When `%` is between two numbers, and the first number is divided by the second, `%` produces the remainder:

```
>>> 9 % 5
4
```

Here's how to get both the (truncated) quotient and remainder at once:

```
>>> divmod(9,5)
(1, 4)
```

Otherwise, you could have calculated them separately:

```
>>> 9 // 5
1
>>> 9 % 5
4
```

You just saw some new concepts here: a *function* named `divmod` is given the integer arguments `9` and `5` and returns a two-item *tuple*. As I mentioned earlier, tuples will take a bow in Chapter 8; functions debut in Chapter 10.

One last math feature is exponentiation with two asterisks (`**`), which also lets you mix integers and floats:

```
>>> 2**3
8
>>> 2.0 ** 3
8.0
>>> 2 ** 3.0
8.0
>>> 0 ** 3
0
```

Precedence

What would you get if you typed the following?

```
>>> 2 + 3 * 4
```

If you do the addition first, `2 + 3` is `5`, and `5 * 4` is `20`. But if you do the multiplication first, `3 * 4` is `12`, and `2 + 12` is `14`. In Python, as in most languages, multiplication has higher *precedence* than addition, so the second version is what you'd see:

```
>>> 2 + 3 * 4
14
```

How do you know the precedence rules? You can look them up, but it's much easier to add parentheses to group your code, showing how you intend the calculation to be carried out:

```
>>> 2 + (3 * 4)
14
```

This example with exponents

```
>>> -5 ** 2
-25
```

is the same as

```
>>> - (5 ** 2)
-25
```

and probably not what you wanted. Parentheses make the intention clear:

```
>>> (-5) ** 2
25
```

This way, anyone reading the code doesn't need to guess its intent or look up precedence rules.

Bases

Integers are assumed to be decimal (base 10) unless you use a prefix to specify another *base*. You might never need to use these other bases, but you'll probably see them in Python code somewhere, sometime.

Most of us have 10 fingers and 10 toes, so we count `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`. Next, we run out of single digits and carry the one to the "ten's place" and put a `0` in the one's place: `10` means "1 ten and 0 ones."[1] Then, it's `11`, `12`, up to `19`, carry the one to make `20` (2 tens and 0 ones), and so on.

A base is the number of digits you can use until you need to "carry the one." In base 2 (`binary`), the only digits are `0` and `1`. This is the famous *bit*. The `0` is the same as a plain old decimal 0, and `1` is the same as a decimal 1. However, in base 2, if you add a `1` to a `1`, you get `10` (1 decimal two plus 0 decimal ones).

In Python, you can express literal integers in three bases besides decimal with these integer prefixes:

- `0b` or `0B` for *binary* (base 2)
- `0o` or `0O` for *octal* (base 8)
- `0x` or `0X` for *hex* (base 16)

These bases are all powers of two and are handy in some cases, although you may never need to use anything other than good old decimal integers.

The interpreter prints these for you as decimal integers. Let's try each of these bases. First, a plain old decimal `10`, which means *1 ten and 0 ones*:

```
>>> 10
10
```

Now, a binary (base two) `0b10`, which means *1 (decimal) two and 0 ones*:

```
>>> 0b10
2
```

Octal (base 8) `0o10` stands for *1 (decimal) eight and 0 ones*:

```
>>> 0o10
8
```

1 Unlike Roman numerals, Arabic numbers don't have a single character that represents *10*.

Hexadecimal (base 16) `0x10` means *1 (decimal) sixteen and 0 ones*:

```
>>> 0x10
16
```

You can go in the other direction, converting an integer to a string with any of these bases:

```
>>> value = 65
>>> bin(value)
'0b1000001'
>>> oct(value)
'0o101'
>>> hex(value)
'0x41'
```

The `chr()` function converts an integer to its single-character string equivalent:

```
>>> chr(65)
'A'
```

And `ord()` goes the other way:

```
>>> ord('A')
65
```

In case you're wondering what "digits" base 16 uses, they are `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, and `f`. The `0xa` is a decimal `10`, and `0xf` is a decimal `15`. Add 1 to `0xf` and you get `0x10` (decimal 16).

Why use different bases from 10? They're useful in *bit-level* operations, which are described in Chapter 20, along with more details about converting numbers from one base to another.

Cats normally have five digits on each forepaw and four on each hindpaw, for a total of 18. If you ever encounter cat scientists in their lab coats, they're often discussing base-18 arithmetic. My cat Chester, seen lounging about in Figure 3-1, was a *polydactyl*, giving him a total of 22 or so (they're hard to distinguish) toes. If he wanted to use all of them to count food fragments surrounding his bowl, he would have likely used a base-22 system (hereafter, the *chesterdigital* system), using `0` through `9` and `a` through `l`.

Figure 3-1. Chester—a fine, furry fellow and inventor of the chesterdigital system

Type Conversions

To change other Python data types to an integer, use `int()`. This function takes one input argument and returns one value, the integer-ized equivalent of the input argument. This will keep the whole number and discard any fractional part.

As you saw at the start of this chapter, Python's simplest data type is the Boolean, which has only the values `True` and `False`. When converted to integers, they represent the values `1` and `0`:

```
>>> int(True)
1
>>> int(False)
0
```

Turning this around, the `bool()` function returns the Boolean equivalent of an integer:

```
>>> bool(1)
True
>>> bool(0)
False
```

Converting a floating-point number to an integer lops off everything after the decimal point:

```
>>> int(98.6)
98
>>> int(1.0e4)
10000
```

Converting a float to a Boolean is no surprise:

```
>>> bool(1.0)
True
>>> bool(0.0)
False
```

Finally, here's an example of getting the integer value from a text string (Chapter 4) that contains only digits, possibly with _ digit separators or an initial + or - sign:

```
>>> int('99')
99
>>> int('-23')
-23
>>> int('+12')
12
>>> int('1_000_000')
1000000
```

If the string represents a nondecimal integer, you can include the base:

```
>>> int('10', 2) # binary
2
>>> int('10', 8) # octal
8
>>> int('10', 16) # hexadecimal
16
>>> int('10', 22) # chesterdigital
22
```

Converting an integer to an integer doesn't change anything but doesn't hurt either:

```
>>> int(12345)
12345
```

If you try to convert something that doesn't look like a number, you'll get an exception:

```
>>> int('99 bottles of beer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '99 bottles of beer'
>>> int('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: ''
```

The preceding text string starts with valid digit characters (`99`), but then keeps on going with others that the `int()` function just won't stand for.

The `int()` function will make integers from floats or strings of digits, but it won't handle strings containing decimal points or exponents:

```
>>> int('98.6')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '98.6'
>>> int('1.0e4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.0e4'
```

If you mix numeric types, Python will sometimes try to automatically convert them for you:

```
>>> 4 + 7.0
11.0
```

The Boolean value `False` is treated as `0` or `0.0` when mixed with integers or floats, and `True` is treated as `1` or `1.0`:

```
>>> True + 2
3
>>> False + 5.0
5.0
```

How Big Is an int?

In old Python 2, the size of an `int` could be limited to 32 or 64 bits, depending on your CPU; 32 bits can store any integer from –2,147,483,648 to 2,147,483,647. A `long` had 64 bits, allowing values from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

In Python 3, the old `long` type is long gone, and an `int` can be *any* size—even greater than 64 bits. You can play with big numbers like a *googol* (one followed by a hundred zeros, named in 1920 by a nine-year-old boy):

```
>>>
>>> googol = 10**100
>>> googol
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000
>>> googol * googol
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

A *googolplex* is `10**googol` (a thousand zeros).[2] This was a suggested name for Google before the company decided on *googol*, but it registered the domain name as *google.com*.

In many languages, trying this would cause *integer overflow*, where the number needs more space than the computer allows, with various bad effects.[3] Python handles googoly integers with no problem.

Floats

Integers are whole numbers, but *floating-point* numbers (called *floats* in Python) have decimal points:

```
>>> 5.
5.0
>>> 5.0
5.0
>>> 05.0
5.0
```

Floats can include a decimal integer exponent after the letter `e`:

```
>>> 5e0
5.0
>>> 5e1
50.0
>>> 5.0e1
50.0
>>> 5.0 * (10 ** 1)
50.0
```

You can use an underscore (_) to separate digits for clarity, as you can for integers:

```
>>> million = 1_000_000.0
>>> million
1000000.0
>>> 1.0_0_1
1.001
```

Floats are handled similarly to integers: you can use the operators (`+`, `-`, `*`, `/`, `//`, `**`, and `%`) and the `divmod()` function.

To convert other types to floats, you use the `float()` function. As before, Booleans act like tiny integers:

```
>>> float(True)
1.0
```

2 Don't type that into Python unless you want to wait for a very long time.

3 Writing into unintended areas of memory is the source of many *buffer overflow* attacks.

```
>>> float(False)
0.0
```

Converting an integer to a float just makes it the proud possessor of a decimal point:

```
>>> float(98)
98.0
>>> float('99')
99.0
```

And you can convert a string containing characters that would be a valid float (digits, signs, decimal point, or an `e` followed by an exponent) to a real float:

```
>>> float('98.6')
98.6
>>> float('-1.5')
-1.5
>>> float('1.0e4')
10000.0
```

All the integer operations that use a symbol before the equals sign work for floats as they do for ints. Here's a floating-point division example, like `a = a / 3`:

```
>>> a = 200.0
>>> a /= 3.0
>>> a
66.66666666666667
```

When you mix integers and floats, Python automatically *promotes* the integer values to float values:

```
>>> 43 + 2.
45.0
```

Python also promotes Booleans to integers or floats:

```
>>> False + 0
0
>>> False + 0.
0.0
>>> True + 0
1
>>> True + 0.
1.0
```

One last point: rounding. Sometimes you don't care about all those numbers after the decimal point. The Python `round()` function is what you need. Give it your number and the number of (rounded) digits to show after the decimal point. If you don't provide the number of digits, `round()` assumes zero and rounds to the nearest whole number:

```
>>> x = 1.2345678
>>> round(x)
1
```

```
>>> round(x, 2)
1.23
>>> x = 2.9876543
>>> round(x)
3
>>> round(x, 2)
2.99
>>>
```

Floats Are Not Exact

Every base will have many common numbers that can't be represented exactly as floating-point values:

```
>>> 1/3
0.3333333333333333
```

(That `3` would go on infinitely. Because computers have only so much space for values, they cut it off at a certain number of bits.)

Therefore, you need to be careful with floats. The official Python documentation has a discussion at "Floating-Point Arithmetic: Issues and Limitations".

The next two sections discuss some solutions.

Fractions

Python has a standard *module* named `fractions` that handles these difficult *rational numbers*. We don't get to modules until Chapter 12 and objects until Chapter 11, so this is a sneak peek. The examples define, in various ways, a `Fraction` object that stores and displays a numerator and denominator. This is a more exact way of defining some numbers than as floats:

```
>>> from fractions import Fraction
>>> Fraction(97, 1)
Fraction(97, 1)
>>> Fraction('97')
Fraction(97, 1)
>>> Fraction(97.0)
Fraction(97, 1)
>>> Fraction('97.0')
Fraction(97, 1)
>>> Fraction(1, 2) + Fraction(7, 16)
Fraction(15, 16)
```

Read "fractions—Rational Numbers" at *python.org* for details.

Decimals

> *If it's a penny for your thoughts and you put in your two cents' worth, then someone, somewhere, is making a penny.*
>
> —Steven Wright

Another standard Python module called `decimal` handles exact representation of decimal numbers. This is useful for Python accounting applications. Again, check the official `decimal` documentation.

If you tried to add 10 cents and 20 cents with floats, you'd get a little extra something:

```
>>> 0.10 + 0.20
0.30000000000000004
```

A common solution is to multiply currency values to get more exact arithmetic. For example, represent dollar or euro amounts as integer pennies (multiply by 100), calculate with integer pennies as much as possible, and convert back to dollars or euros when done.

Math Functions

Python supports complex numbers and has the usual math functions such as square roots, cosines, and so on. Let's save them for Chapter 25, in which we also discuss using Python in scientific contexts. If you're curious now, I'll again point you to the standard Python math documentation.

Review/Preview

This chapter showed how to represent Booleans and integers as well as floating-point numbers, and you learned some of the operations you can do with them.

Next: moving up to the exciting world of text strings.

Practice

This chapter introduced the atoms of Python: numbers, Booleans, and variables. Let's try a few small exercises with them in the interactive interpreter. Try these exercises before looking up the answers in the Appendix.

3.1 How many seconds are in an hour? Use the interactive interpreter as a calculator and multiply the number of seconds in a minute (`60`) by the number of minutes in an hour (also `60`).

3.2 Assign the result from the previous task (seconds in an hour) to a variable called `seconds_per_hour`.

3.3 How many seconds are in a day? Use your `seconds_per_hour` variable.

3.4 Calculate seconds per day again, but this time save the result in a variable called `seconds_per_day`.

3.5 Divide `seconds_per_day` by `seconds_per_hour`. Use floating-point (`/`) division.

3.6 Divide `seconds_per_day` by `seconds_per_hour`, using integer (`//`) division. Did this number agree with the floating-point value from the previous question, aside from the final `.0`?

CHAPTER 4
Strings

I always liked strange characters.
—Tim Burton

Computer books often give the impression that programming is all about math. Actually, most programmers work with strings of text more often than numbers. Logical (and creative!) thinking is often more important than math skills.

Strings are our first example of a Python sequence. A *sequence* is an ordered collection of elements. Some of the information that you can get from any sequence are as follows:

- Whether an element is in the sequence
- The index of an element by its value
- The element at a particular index
- A *slice* of elements in a given range
- The length of the sequence
- The minimum and maximum element values

Other sequences that you'll see in coming chapters include tuples and lists (Chapter 8) and bytes and bytearrays (Chapter 5).

A Python string is a sequence of *characters*. But what's a character? It's the smallest unit in a writing system, and includes letters, digits, symbols, punctuation, and even whitespace or directives like line feeds. A character is defined by its meaning (how it's used), not how it looks. It can have more than one visual representation (in different fonts), and more than one character can have the same appearance (such as the visual `H`, which means the `H` sound in the Latin alphabet but the Latin `N` sound in Cyrillic).

This chapter concentrates on making and formatting simple text strings, using ASCII (basic character set) examples. Two important text topics are deferred to Chapter 17: Unicode characters (like the `H` and `N` issue I just mentioned) and regular expressions (pattern matching).

Unlike other languages, strings in Python are *immutable*. You can't change a string in place, but you can copy parts of strings to a new string to get the same effect. We look at how to do this shortly.

Single characters don't exist on their own in Python. They're always part of a string. A string can have one character, a million, or even none (an *empty string*).

Create with Quotes

You make a Python string by enclosing characters in matching single or double quotation marks:

```
>>> 'Snap'
'Snap'
>>> "Crackle"
'Crackle'
>>> 'Flop'
'Flop'
```

The interactive interpreter echoes strings with a single quote, but all are treated exactly the same by Python.

Python has a few special types of strings, indicated by a letter before the first quote:

- `f` or `F` starts an *f-string*, used for formatting, and described near the end of this chapter.
- `r` or `R` starts a *raw string*, used to prevent *escape sequences* in the string (see "Escape with \" on page 53 and Chapter 19 for its use in string pattern matching).
- Then, there's the combination `fr` (or `FR`, `Fr`, or `fR`) that starts a raw f-string.
- A `u` starts a Unicode string, which is the same as a plain string.
- And a `b` starts a value of type `bytes` (Chapter 5).

Unless I mention one of these special types, I'm always talking about plain old Python text strings. Each character in a string can be anything defined in the Unicode standard. Again, I'm holding off a big discussion of Unicode until Chapter 17, but the main point to know for now is that Unicode is *much* bigger than ASCII.

Why have two kinds of quote characters? The main purpose is to create strings containing quote characters. You can have single quotes inside double-quoted strings, or double quotes inside single-quoted strings:

```
>>> "'Nay!' said the naysayer. 'Neigh?' said the horse."
"'Nay!' said the naysayer. 'Neigh?' said the horse."
>>> 'The rare double quote in captivity: ".'
'The rare double quote in captivity: ".'
>>> 'A "two by four" is actually 1 1/2" × 3 1/2".'
'A "two by four" is actually 1 1/2" × 3 1/2".'
>>> "'There's the man that shot my paw!' cried the limping hound."
"'There's the man that shot my paw!' cried the limping hound."
```

You can also use three single quotes (`'''`) or three double quotes (`"""`):

```
>>> '''Boom!'''
'Boom'
>>> """Eek!"""
'Eek!'
```

Triple quotes aren't very useful for short strings like these. Their most common use is to create *multiline strings*, like this classic poem from Edward Lear:

```
>>> poem = '''There was a Young Lady of Norway,
... Who casually sat in a doorway;
... When the door squeezed her flat,
... She exclaimed, "What of that?"
... This courageous Young Lady of Norway.'''
>>>
```

This was entered in the interactive interpreter, which prompted us with `>>>` for the first line and continuation prompts `...` until we entered the final triple quotes and went to the next line.

If you tried to create that poem without triple quotes, Python would make a fuss when you went to the second line:

```
>>> poem = 'There was a young lady of Norway,
  File "<stdin>", line 1
    poem = 'There was a young lady of Norway,
                                            ^
SyntaxError: EOL while scanning string literal
>>>
```

(`EOL` means an end-of-line exception.)

If you have multiple lines within triple quotes, the line-ending characters will be preserved in the string. If you have leading or trailing spaces, they'll also be kept:

```
>>> poem2 = '''I do not like thee, Doctor Fell.
...     The reason why, I cannot tell.
...     But this I know, and know full well:
...     I do not like thee, Doctor Fell.
```

```
... '''
>>> print(poem2)
I do not like thee, Doctor Fell.
    The reason why, I cannot tell.
    But this I know, and know full well:
    I do not like thee, Doctor Fell.

>>>
```

By the way, there's a difference between the output of `print()` and the automatic echoing done by the interactive interpreter:

```
>>> poem2
'I do not like thee, Doctor Fell.\n    The reason why, I cannot tell.\n    But
this I know, and know full well:\n    I do not like thee, Doctor Fell.\n'
```

The `print()` function strips quotes from strings and prints their contents. This function is meant for human output. It helpfully adds a space between each of the elements it prints, and a newline at the end:

```
>>> print('Give', "us", '''some''', """space""")
Give us some space
```

If you don't want the space or newline, Chapter 17 explains how to avoid them.

The interactive interpreter prints the string with individual quotes and escape characters such as `\n`, which are explained in "Escape with \" on page 53:

```
>>> """'Guten Morgen, mein Herr!'
... said mad king Ludwig to his wig."""
"'Guten Morgen, mein Herr!'\nsaid mad king Ludwig to his wig."
```

Finally, an *empty string* has no characters at all but is perfectly valid. You can create an empty string with any of the aforementioned quotes:

```
>>> ''
''
>>> ""
''
>>> ''''''
''
>>> """""""
''
>>>
```

Create with str()

You can make a string from another data type by using the `str()` function:

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
```

```
>>> str(True)
'True'
```

Python uses `str()` internally when you call `print()` with objects that are not strings and when doing string formatting, which you'll see later in this chapter.

Escape with \

Python lets you *escape* the meaning of some characters within strings to achieve effects that would otherwise be difficult to express. By preceding a character with a backslash (\), you give it a special meaning. The most common escape sequence is \n, which means to begin a new line. With this, you can create multiline strings from a one-line string:

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
```

You may see the escape sequence \t (tab) used to align text:

```
>>> print('\tabc')
        abc
>>> print('a\tbc')
a       bc
>>> print('ab\tc')
ab      c
>>> print('abc\t')
abc
```

(The final string has a terminating tab which, of course, you can't see.)

You might also need \' or \" to specify a literal single or double quote inside a string that's quoted by the same character:

```
>>> testimony = "\"I did nothing!\" he said. \"Or that other thing.\""
>>> testimony
'"I did nothing!" he said. "Or that other thing."'
>>> print(testimony)
"I did nothing!" he said. "Or that other thing."
```

```
>>> fact = "The world's largest rubber duck was 54'2\" by 65'7\" by 105'"
>>> print(fact)
The world's largest rubber duck was 54'2" by 65'7" by 105'
```

And if you need a literal backslash, type two of them (the first escapes the second):

```
>>> speech = 'The backslash (\\) bends over backwards to please you.'
>>> print(speech)
The backslash (\) bends over backwards to please you.
>>>
```

As I mentioned early in this chapter, a *raw string* negates these escapes:

```
>>> info = r'Type a \n to get a new line in a normal string'
>>> info
'Type a \\n to get a new line in a normal string'
>>> print(info)
Type a \n to get a new line in a normal string
```

(The extra backslash in the first `info` output was added by the interactive interpreter.)

A raw string does not undo any real (not `\n`) newlines:

```
>>> poem = r'''Boys and girls, come out to play.
... The moon doth shine as bright as day.'''
>>> poem
'Boys and girls, come out to play.\nThe moon doth shine as bright as day.'
>>> print(poem)
Boys and girls, come out to play.
The moon doth shine as bright as day.
```

Combine with +

You can combine literal strings or string variables in Python by using the + operator:

```
>>> 'Release the kraken! ' + 'No, wait!'
'Release the kraken! No, wait!'
```

You can also combine *literal strings* (not string variables) by having one after the other:

```
>>> "My word! " "A gentleman caller!"
'My word! A gentleman caller!'
>>> "Alas! ""The kraken!"
'Alas! The kraken!'
```

If you have a lot of these, you can avoid escaping the line endings by surrounding them with parentheses:

```
>>> vowels = ( 'a'
... "e" '''i'''
... 'o' """u"""
... )
>>> vowels
'aeiou'
```

Python does *not* add spaces for you when concatenating strings, so in some earlier examples, we needed to explicitly include spaces. Python *does* add a space between each argument to a `print()` statement and a newline at the end:

```
>>> a = 'Duck.'
>>> b = a
>>> c = 'Grey Duck!'
>>> a + b + c
```

```
'Duck.Duck.Grey Duck!'
>>> print(a, b, c)
Duck. Duck. Grey Duck!
```

Duplicate with *

You use the * operator to duplicate a string. Try typing these lines into your interactive interpreter and see what they print:

```
>>> start = 'Na ' * 4 + '\n'
>>> middle = 'Hey ' * 3 + '\n'
>>> end = 'Goodbye.'
>>> print(start + start + middle + end)
```

Notice that the * has higher precedence than +, so the string is duplicated before the line feed is tacked on.

Get a Character by [offset]

To get a single character from a string, specify its *offset* inside square brackets after the string's name. The first (leftmost) offset is 0, the next is 1, and so on. The last (rightmost) offset can be specified with –1 so you don't have to count; going to the left are –2, –3, and so on:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[0]
'a'
>>> letters[1]
'b'
>>> letters[-1]
'z'
>>> letters[-2]
'y'
>>> letters[25]
'z'
>>> letters[5]
'f'
```

If you specify an offset that is the length of the string or longer (remember, offsets go from 0 to length – 1), you'll get an exception:

```
>>> letters[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Indexing works the same with the other sequence types (lists and tuples), which I cover in Chapter 8.

Because strings are immutable, you can't insert a character directly into one or change the character at a specific index. Let's try to change `Henny` to `Penny` and see what happens:

```
>>> name = 'Henny'
>>> name[0] = 'P'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Instead, you need to use a combination of string functions such as `replace()` or a slice (which we look at in a moment):

```
>>> name = 'Henny'
>>> name.replace('H', 'P')
'Penny'
>>> 'P' + name[1:]
'Penny'
```

We didn't change the value of `name`. The interactive interpreter just printed the result of the replacement.

Get a Substring with a Slice

You can extract a *substring* (a part of a string) from a string by using a *slice*. You define a slice by using square brackets, a *`start`* offset, an *`end`* offset, and an optional *`step`* count between them. You can omit some of these. The slice will include characters from offset *`start`* to one before *`end`*:

- `[:]` extracts the entire sequence from start to end.
- `[` *`start`* `:]` specifies from the *`start`* offset to the end.
- `[:` *`end`* `]` specifies from the beginning to the *`end`* offset minus 1.
- `[` *`start`* `:` *`end`* `]` indicates from the *`start`* offset to the *`end`* offset minus 1.
- `[` *`start`* `:` *`end`* `:` *`step`* `]` extracts from the *`start`* offset to the *`end`* offset minus 1, skipping characters by *`step`*.

As before, offsets go 0, 1, and so on from the start to the right, and –1,–2, and so forth from the end to the left. If you don't specify *`start`*, the slice uses 0 (the beginning). If you don't specify *`end`*, it uses the end of the string.

Let's make a string of the lowercase English letters:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
```

Using a plain `:` is the same as `0:` (the entire string):

```
>>> letters[:]
'abcdefghijklmnopqrstuvwxyz'
```

Here's an example from offset 20 to the end:

```
>>> letters[20:]
'uvwxyz'
```

Now, from offset 10 to the end:

```
>>> letters[10:]
'klmnopqrstuvwxyz'
```

And another, offset 12 through 14. Python does not include the end offset in the slice. The start offset is *inclusive*, and the end offset is *exclusive*:

```
>>> letters[12:15]
'mno'
```

Access the three last characters:

```
>>> letters[-3:]
'xyz'
```

In this next example, we go from offset 18 to the fourth before the end; notice the difference from the previous example, in which starting at –3 gets the `x`, but ending at –3 actually stops at –4, the `w`:

```
>>> letters[18:-3]
'stuvw'
```

In the following, we extract from 6 before the end, to 3 before the end:

```
>>> letters[-6:-2]
'uvwx'
```

If you want a step size other than 1, specify it after a second colon, as shown in the next series of examples.

From the start to the end, in steps of 7 characters:

```
>>> letters[::7]
'ahov'
```

From offset 4 to 19, by 3:

```
>>> letters[4:20:3]
'ehknqt'
```

From offset 19 to the end, by 4:

```
>>> letters[19::4]
'tx'
```

From the start to offset 20, by 5:

```
>>> letters[:21:5]
'afkpu'
```

(Again, the *end* needs to be one more than the actual offset.)

And that's not all! Given a negative step size, this handy Python slicer can also step backward. This starts at the end and ends at the start, skipping nothing:

```
>>> letters[-1::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

It turns out that you can get the same result by using this:

```
>>> letters[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

Slices are more forgiving of bad offsets than are single-index lookups with `[]`. A slice offset earlier than the beginning of a string is treated as `0`, and one after the end is treated as `-1`, as is demonstrated in this next series of examples.

From 50 before the end, to the end:

```
>>> letters[-50:]
'abcdefghijklmnopqrstuvwxyz'
```

From 51 before the end, to 50 before the end:

```
>>> letters[-51:-50]
''
```

From the start, to 69 after the start:

```
>>> letters[:70]
'abcdefghijklmnopqrstuvwxyz'
```

From 70 after the start, to 70 after the start:

```
>>> letters[70:71]
''
```

Get Length with len()

So far, we've used special punctuation characters such as `+` to manipulate strings. But Python has only so many of these. Now let's begin to use some of Python's built-in *functions*: named pieces of code that perform certain operations. They get more coverage in Chapter 10.

The `len()` function counts characters in a string:

```
>>> len(letters)
26
>>> empty = ""
>>> len(empty)
0
```

You can use `len()` with other sequence types too, as you'll see in Chapter 8.

Split with split()

Unlike `len()`, some functions are specific to strings. To use a string function, type the name of the string variable, a dot, the name of the function, and any *arguments* that the function needs: *`string.function(arguments)`*.

You can use the built-in `split()` function to break a string into a list of smaller strings based on a *separator*. Again, we look at lists in Chapter 8. A *list* is a sequence of values, separated by commas and surrounded by square brackets:

```
>>> tasks = 'get gloves,get mask,give cat vitamins,call ambulance'
>>> tasks.split(',')
['get gloves', 'get mask', 'give cat vitamins', 'call ambulance']
```

In the preceding example, the string is called `tasks`, and the string function is called `split()`, with the single separator argument `,`. If you don't specify a separator, `split()` uses any sequence of whitespace characters—newlines, spaces, and tabs:

```
>>> tasks.split()
['get', 'gloves,get', 'mask,give', 'cat', 'vitamins,call', 'ambulance']
```

You still need the parentheses when calling `split()` with no arguments; that's how Python knows you're calling a function.

Combine with join()

Not too surprisingly, the `join()` function is the opposite of `split()`: it collapses a list of strings into a single string. The `join()` function looks a bit backward because you specify the string that glues everything together first and then the list of strings to glue: *`string`* `.join(` *`list`* `)`. So, to join the list `lines` with separating newlines, you would write `'\n'.join(lines)`. As you'll see in Chapter 8, one way to make a list is with square brackets (`[` and `]`) surrounding a comma-separated sequence of items. In the following example, let's join some names in a list with a comma and a space:

```
>>> crypto_list = ['Yeti', 'Bigfoot', 'Loch Ness Monster']
>>> crypto_string = ', '.join(crypto_list)
>>> print('Found and signing book deals:', crypto_string)
Found and signing book deals: Yeti, Bigfoot, Loch Ness Monster
```

Substitute with replace()

You use `replace()` for simple substring substitution. Give it the old substring, the new one, and the number of instances of the old substring to replace. The function returns the changed string but does not modify the original. If you omit this final

count argument, `join()` replaces all instances. In this example, only one string (`duck`) is matched and replaced in the returned string:

```
>>> setup = "a duck goes into a bar..."
>>> setup.replace('duck', 'marmoset')
'a marmoset goes into a bar...'
>>> setup
'a duck goes into a bar...'
```

Change up to 100 of them:

```
>>> setup.replace('a ', 'a famous ', 100)
'a famous duck goes into a famous bar...'
```

When you know the exact substring(s) you want to change, `replace()` is a good choice. But watch out. In the second example, if we had substituted for the single-character string `'a'` rather than the two-character string `'a '` (`a` followed by a space), we would have also changed `a` in the middle of other words:

```
>>> setup.replace('a', 'a famous', 100)
'a famous duck goes into a famous ba famousr...'
```

Sometimes you want to ensure that the substring is a whole word, or the beginning of a word, and so on. In those cases, you need regular expressions, which are described in numbing detail in Chapter 17.

Work with Prefixes and Suffixes

You may need to find, change, or delete the prefix or suffix of a string. If this never happens to you, feel free to make yourself a sandwich now. Otherwise, here are a few useful Python string methods:

```
>>> s = "inconceivable"
>>> s.startswith("in")
True
>>> s.startswith("un")
False
>>> s.endswith("able")
True
>>> s.endswith("abominable")
False
>>> s.removeprefix("in")
'conceivable'
>>> s.removesuffix("conceivable")
'in'
```

The `removeprefix()` and `removesuffix()` functions were added in Python 3.9.

To add a prefix or suffix, use + to concatenate (join) the old word and the new part:

```
>>> "ultra" + s
'ultrainconceivable'
>>> s + "ness"
'inconceivableness'
```

Strip with strip()

It's common to strip leading or trailing "padding" characters from a string, especially spaces. The `strip()` functions shown here assume that you want to get rid of whitespace characters (`' '`, `'\t'`, `'\n'`) if you don't give them an argument. The `strip()` function strips both ends, `lstrip()` only from the left, and `rstrip()` only from the right. Let's say the string variable `world` contains the string `earth` floating in spaces:

```
>>> world = "    earth   "
>>> world.strip()
'earth'
>>> world.strip(' ')
'earth'
>>> world.lstrip()
'earth   '
>>> world.rstrip()
'    earth'
```

If the character was not there, nothing happens:

```
>>> world.strip('!')
'    earth   '
```

Besides no argument (meaning whitespace characters) or a single character, you can also tell `strip()` to remove any character in a multicharacter string:

```
>>> blurt = "What the...!!?"
>>> blurt.strip('.?!')
'What the'
```

Here are some character groups that are useful with `strip()`:

```
>>> import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> blurt = "What the...!!?"
>>> blurt.strip(string.punctuation)
'What the'
>>> prospector = "What in tarnation ...??!!"
>>> prospector.strip(string.whitespace + string.punctuation)
'What in tarnation'
```

Search and Select

Python has a large set of string functions. Let's explore how the most common of them work. Our test subject is the following string containing the text of the immortal poem "What Is Liquid?" by Margaret Cavendish, Duchess of Newcastle:

```
>>> poem = '''All that doth flow we cannot liquid name
... Or else would fire and water be the same;
... But that is liquid which is moist and wet
... Fire that property can never get.
... Then 'tis not cold that doth the fire put out
... But 'tis the wet that makes it die, no doubt.'''
```

Inspiring!

To begin, get the first 13 characters (offsets 0 to 12):

```
>>> poem[:13]
'All that doth'
```

How many characters are in this poem? (Spaces and newlines are included in the count.)

```
>>> len(poem)
250
```

Does it start with the letters `All`?

```
>>> poem.startswith('All')
True
```

Does it end with `That's all, folks!`?

```
>>> poem.endswith('That\'s all, folks!')
False
```

Python has two methods, `find()` and `index()`, for finding the offset of a substring, and has two versions of each (starting from the beginning or the end). They work the same if the substring is found. If it isn't, `find()` returns `-1`, and `index()` raises an exception.

Let's find the offset of the first occurrence of the word `the` in the poem:

```
>>> word = 'the'
>>> poem.find(word)
73
>>> poem.index(word)
73
```

And the offset of the last `the`:

```
>>> word = 'the'
>>> poem.rfind(word)
214
```

```
>>> poem.rindex(word)
214
```

Change case if the substring isn't in there:

```
>>> word = "duck"
>>> poem.find(word)
-1
>>> poem.rfind(word)
-1
>>> poem.index(word)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> poem.rfind(word)
-1
>>> poem.rindex(word)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

How many times does the three-letter sequence `the` occur?

```
>>> word = 'the'
>>> poem.count(word)
3
```

Are all the characters in the poem either letters or numbers?

```
>>> poem.isalnum()
False
```

Nope, the poem has some punctuation characters.

Change Case

In this section, we look at more uses of the built-in string functions. Our test string is again the following:

```
>>> setup = 'a duck goes into a bar...'
```

Remove `.` sequences from both ends:

```
>>> setup.strip('.')
'a duck goes into a bar'
```

Because strings are immutable, none of these examples changes the `setup` string. Each example takes the value of `setup`, does something to it, and returns the result as a new string.

Capitalize the first word:

```
>>> setup.capitalize()
'A duck goes into a bar...'
```

Capitalize all the words:

```
>>> setup.title()
'A Duck Goes Into A Bar...'
```

Convert all characters to uppercase:

```
>>> setup.upper()
'A DUCK GOES INTO A BAR...'
```

Convert all characters to lowercase:

```
>>> setup.lower()
'a duck goes into a bar...'
```

Swap uppercase and lowercase:

```
>>> setup.swapcase()
'A DUCK GOES INTO A BAR...'
```

Set Alignment

Now, let's work with some layout alignment functions. The string is aligned within the specified total number of spaces (`30` here).

Center the string within 30 spaces:

```
>>> setup.center(30)
'  a duck goes into a bar...   '
```

Left-justify:

```
>>> setup.ljust(30)
'a duck goes into a bar...     '
```

Right-justify:

```
>>> setup.rjust(30)
'     a duck goes into a bar...'
```

Next, we look at more ways to format a string.

Apply Formatting

You've seen that you can *concatenate* strings by using +. Let's look at how to *interpolate* data values into strings using various formats. You can use this approach to produce reports, forms, and other outputs that need appearances to be just so.

Besides the functions in the preceding section, Python has three ways of formatting strings:

- Old style (supported in Python 2 and 3)
- New style (Python 2.6 and up)
- F-strings (Python 3.6 and up)

I prefer using f-strings, but you'll see the older styles and should at least understand them.

Old style: %

The old style of string formatting has the form *`format_string % data`*. Inside the format string are interpolation sequences. Table 4-1 illustrates that the simplest sequence is a `%` followed by a letter indicating the data type to be formatted.

Table 4-1. Conversion types

Format code	Usage
`%s`	String
`%d`	Decimal integer
`%x`	Hex integer
`%o`	Octal integer
`%f`	Decimal float
`%e`	Exponential float
`%g`	Decimal or exponential float
`%%`	A literal %

You can use a `%s` for any data type, and Python will format it as a string with no extra spaces.

Following are some simple examples. First, format an integer with different bases:

```
>>> '%s' % 42
'42'
>>> '%d' % 42
'42'
>>> '%x' % 42
'2a'
>>> '%o' % 42
'52'
```

Now format a float, with different styles:

```
>>> '%s' % 7.03
'7.03'
```

```
>>> '%f' % 7.03
'7.030000'
>>> '%e' % 7.03
'7.030000e+00'
>>> '%g' % 7.03
'7.03'
```

Next, format an integer and a literal %:

```
>>> '%d%%' % 100
'100%'
```

Let's try some string and integer interpolation:

```
>>> cat = 'Chester'
>>> weight = 28
>>> "My cat %s weighs %s pounds" % (cat, weight)
'My cat Chester weighs 28 pounds'
```

That `%s` inside the string means to interpolate a string. The number of `%` appearances in the string needs to match the number of data items after the `%` that follows the string. A single data item goes right after that final `%`. Multiple pieces of data must be grouped into a *tuple* (details in Chapter 8; it's bounded by parentheses, separated by commas) such as `(cat, weight)`.

Even though `weight` is an integer, the `%s` inside the string converts it to a string.

You can add other values in the format string between the `%` and the type specifier to designate minimum and maximum widths, alignment, and character filling. This is a little language in its own right, and more limited than the one in the next two sections. Let's take a quick look at these values:

- An initial `%` character.
- An optional *alignment* character: nothing or `+` means right-align, and `-` means left-align.
- An optional *minwidth* field width to use.
- An optional `.` character to separate *minwidth* and *maxchars*.
- An optional *maxchars* (if conversion type is `s`) indicating the number of characters to print from the data value. If the conversion type is `f`, this specifies *precision* (the number of digits to print after the decimal point).
- The *conversion type* character from the earlier table.

This is confusing, so here are some examples for a string:

```
>>> thing = 'woodchuck'
>>> '%s' % thing
'woodchuck'
>>> '%12s' % thing
```

```
'   woodchuck'
>>> '%+12s' % thing
'   woodchuck'
>>> '%-12s' % thing
'woodchuck   '
>>> '%.3s' % thing
'woo'
>>> '%12.3s' % thing
'         woo'
>>> '%-12.3s' % thing
'woo         '
```

Once more with feeling, and a float with `%f` variants:

```
>>> thing = 98.6
>>> '%f' % thing
'98.600000'
>>> '%12f' % thing
'   98.600000'
>>> '%+12f' % thing
'  +98.600000'
>>> '%-12f' % thing
'98.600000   '
>>> '%.3f' % thing
'98.600'
>>> '%12.3f' % thing
'      98.600'
>>> '%-12.3f' % thing
'98.600      '
```

And an integer with `%d`:

```
>>> thing = 9876
>>> '%d' % thing
'9876'
>>> '%12d' % thing
'        9876'
>>> '%+12d' % thing
'       +9876'
>>> '%-12d' % thing
'9876        '
>>> '%.3d' % thing
'9876'
>>> '%12.3d' % thing
'        9876'
>>> '%-12.3d' % thing
'9876        '
```

For an integer, the `%+12d` forces the sign to be printed, and the format strings with `.3` in them have no effect as they do for a float.

New style: {} and format()

"New style" formatting has the form *format_string*.format(*data*).

The format string is not exactly the same as the old style described in the preceding section. The simplest usage is demonstrated here:

```
>>> thing = 'woodchuck'
>>> '{}'.format(thing)
'woodchuck'
```

The arguments to the `format()` function need to be in the same order as the {} placeholders in the format string:

```
>>> thing = 'woodchuck'
>>> place = 'lake'
>>> 'The {} is in the {}.'.format(thing, place)
'The woodchuck is in the lake.'
```

With new-style formatting, you can also specify the arguments by position like this:

```
>>> 'The {1} is in the {0}.'.format(place, thing)
'The woodchuck is in the lake.'
```

The value `0` refers to the first argument, `place`, and `1` refers to `thing`.

The arguments to `format()` can also be named arguments:

```
>>> 'The {thing} is in the {place}'.format(thing='duck', place='bathtub')
'The duck is in the bathtub'
```

or a dictionary:

```
>>> d = {'thing': 'duck', 'place': 'bathtub'}
```

In the following example, `{0}` is the first argument to `format()` (the dictionary `d`):

```
>>> 'The {0[thing]} is in the {0[place]}.'.format(d)
'The duck is in the bathtub.'
```

These examples all print their arguments with default formats. New-style formatting has a slightly different format string definition from the old-style one (examples follow):

- An initial colon (`:`).
- An optional *fill* character (the default is `' '`) to pad the value string if it's shorter than *minwidth*.
- An optional *alignment* character. This time, left alignment is the default. The `<` character also means *left*, `>` means *right*, and `^` means *center*.

- An optional *sign* for numbers. Nothing means prepend a minus sign (`-`) only for negative numbers. Using `' '` means prepend a minus sign for negative numbers, and a space (`' '`) for positive ones.
- An optional *minwidth*.
- An optional period (`.`) to separate *minwidth* and *maxchars*.
- An optional *maxchars*.
- The *conversion type*.

The following shows various ways of formatting a string with two string variables:

```
>>> thing = 'wraith'
>>> place = 'window'
>>> 'The {} is at the {}'.format(thing, place)
'The wraith is at the window'
>>> 'The {:10s} is at the {:10s}'.format(thing, place)
'The wraith     is at the window    '
>>> 'The {:<10s} is at the {:<10s}'.format(thing, place)
'The wraith     is at the window    '
>>> 'The {:^10s} is at the {:^10s}'.format(thing, place)
'The   wraith   is at the   window  '
>>> 'The {:>10s} is at the {:>10s}'.format(thing, place)
'The     wraith is at the     window'
>>> 'The {:!^10s} is at the {:!^10s}'.format(thing, place)
'The !!wraith!! is at the !!window!!'
```

Newest Style: f-strings

F-strings appeared in Python 3.6 and are now the recommended way of formatting strings.

To make an f-string, do the following:

- Type the letter `f` or `F` directly before the initial single- or triple-quote characters.
- Include variable names or expressions within curly braces (`{}`) to get their values interpolated into the string.

This approach is like the previous section's new-style formatting, but without the `format()` function, and without empty brackets (`{}`) or positional ones (`{1}`) in the format string. F-strings use curly braces (`{}`) differently:

```
>>> thing = 'wereduck'
>>> place = 'werepond'
>>> f'The {thing} is in the {place}'
'The wereduck is in the werepond'
```

As I already mentioned, expressions are also allowed inside the curly braces:

```
>>> f'The {thing.capitalize()} is in the {place.rjust(20)}'
'The Wereduck is in the             werepond'
```

This means that the things that you could do inside `format()` in the previous section, you can now do inside a `{}` in your main string. This seems easier to read.

F-strings use the same formatting language (width, padding, alignment) as new-style formatting, after a colon (`:`):

```
>>> f'The {thing:>20} is in the {place:.^20}'
'The             wereduck is in the ......werepond......'
```

Starting in Python 3.8, f-strings gain a new shortcut that's helpful when you want to print variable names as well as their values. This is handy when debugging. The trick is to have a single `=` after the name in the `{}`-enclosed part of the f-string:

```
>>> f'{thing =}, {place =}'
thing = 'wereduck', place = 'werepond'
```

The name can be an expression, and it will be printed literally:

```
>>> f'{thing[-4:] =}, {place.title() =}'
thing[-4:] = 'duck', place.title() = 'Werepond'
```

Finally, the `=` can be followed by a `:` and the formatting arguments like width and alignment:

```
>>> f'{thing = :>4.4}'
thing = 'were'
```

More String Things

Python has many more string functions than I've shown here. Some will turn up in later chapters (especially Chapter 17), but you can find all the details in the standard documentation.

Review/Preview

Strings are a strong component of the Python language. In usual daily programming jobs, you'll probably spend a lot of time messing with strings. They're an example of a sequence and include methods to access characters by their position or pattern.

Because strings are immutable, you can't change the contents of one after it's been defined. But you can use slices to copy out the characters that you want, and combine them with others to make a new string. Say you want to grab the first two and last two characters of an existing string, and add a `!` in the middle (people have done stranger things):

```
>>> s1 = "abcdef"
>>> s2 = s1[0:2] + "!" + s1[-2:]
>>> s2
'ab!ef'
```

Coming next: the binary counterparts of text strings: `bytes` and `bytearray`.

Practice

4.1 Capitalize the word starting with `m`:

```
>>> song = """When an eel grabs your arm,
... And it causes great harm,
... That's - a moray!"""
```

4.2 Write the following poem by using old-style formatting. Substitute the strings `roast beef`, `ham`, `head`, and `clam` into this string:

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
```

4.3 Write a form letter by using new-style formatting. Save the following string as `letter` (you'll use it in the next exercise):

```
Dear {salutation} {name},

Thank you for your letter. We are sorry that our {product}
{verbed} in your {room}. Please note that it should never
be used in a {room}, especially near any {animals}.

Send us your receipt and {amount} for shipping and handling.
We will send you another {product} that, in our tests,
is {percent}% less likely to have {verbed}.

Thank you for your support.

Sincerely,
{spokesman}
{job_title}
```

4.4 Assign values to variable strings named `salutation`, `name`, `product`, `verbed` (past tense verb), `room`, `animals`, `percent`, `spokesman`, and `job_title`. Print `letter` with these values, using `letter.format()`.

4.5 After public polls to name things, a pattern emerged: an English submarine (Boaty McBoatface), an Australian racehorse (Horsey McHorseface), and a Swedish train (Trainy McTrainface). Use `%` formatting to print the winning name at the state fair for a prize duck, gourd, and spitz.

4.6 Do the same, with `format()` formatting.

4.7 Once more, with feeling, and *f-strings*.

CHAPTER 5
Bytes and Bytearray

I'm feeling a bit off.
—Byte calling in sick

Python 3 introduced the following sequences of eight-bit integers, with possible values from 0 to 255, in two types:

- An immutable sequence of 8-bit values: `bytes`
- A mutable sequence of 8-bit values: `bytearray`

This chapter covers the basics of both types. Chapter 20 goes into more detail on their real-life uses, including binary file formats. You'll probably deal with binary data much less often than text strings.

The official Python docs have all the details on these data types. I'm including just the most common and useful in this chapter.

Here's one way to think of bytes versus strings:

- Bytes are like equally sized beads on a wire.
- Strings are like a charm bracelet.

Bytes

Bytes, like strings, are immutable. You can't append, insert, or change the contents of a `bytes` value after you've created it.

Create with Quotes

A literal `bytes` object is surrounded by quotes like a text string, but has a `b` right before the initial quote character. Each byte in it is specified either as an ASCII character or a two-character hex literal:

```
>>> b1 = b'ABC\x01\x02\x03\x41\x42\x43'
>>> b1
b'ABC\x01\x02\x03ABC'
>>> b2 = b'abc\x01\x02\x03\x61\x62\x63'
>>> b2
b'abc\x01\x02\x03abc'
```

This doesn't mean that a `bytes` value can contain text characters. It means that you can *specify* a byte with its hex value *or* its ASCII equivalent, if it has one. Python allows this ASCII shortcut for convenience on input and displaying output.

Create with bytes()

Beginning with a list (a sequence of any kind of values; see Chapter 8) called `blist`, this next example creates a `bytes` variable called `the_bytes`:

```
>>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes
b'\x01\x02\x03\xff'
```

This next example demonstrates that you can't change a `bytes` variable:

```
>>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes[1] = 127
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

In "Bytearray" on page 77, you'll see that you *can* change the innards of a `bytearray`.

Each of these would create a 256-element result, with values from 0 to 255:

```
>>> the_bytes = bytes(range(0, 256))
```

When printing `bytes` or `bytearray` data, Python uses `\x`*hh* for nonprintable bytes[1] and their ASCII equivalents for printable ones (plus some common escape characters, such as `\n` instead of `\x0a`). Here's the printed representation of `the_bytes` (manually reformatted to show 16 bytes per line):

1 Each *h* represents a hex character (0–9, a–f, or A–F).

```
>>> the_bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f
\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~\x7f
\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f
\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf
\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf
\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef
\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff'
```

This can be confusing, because they're bytes (teeny integers), not characters, and Python displays some of them as their ASCII character equivalents.

Create from a Hex String

To convert from a hex string to bytes, each hex value needs to be two hex characters, so you can't use the single-character ASCII equivalents. A hex string can contain the digits `0` through `9`, and the letters `a` through `f`, or `A` through `F`. Python's `fromhex()` convenience function does this conversion, but it's a bit picky about how you enter the hex string:

```
>>> hex_string = "FF 01 A B C DEF"
>>> the_bytes = bytes.fromhex(hex_string)
Traceback (most recent call last):
  File "<python-input-53>", line 1, in <module>
    the_bytes = bytes.fromhex(hex_string)
ValueError: non-hexadecimal number found in fromhex() arg at position 7
```

Using two characters for each, and ignoring spaces, we have this:

```
>>> hex_string = "FF 01 61 62 63 64 6566"
>>> the_bytes = bytes.fromhex(hex_string)
>>> the_bytes
b'\xff\x01abcdef'
```

Notice that we need to say `bytes.fromhex(hexstring)` instead of something like `hex string.bytes()` or plain `fromhex(hexstring)`. This `fromhex()` is a *class method* that's included in the definition of `bytes` objects. Chapter 11 gets into what this means.

Decode and Encode Bytes and Strings

In Chapter 17, we get into the gory details on Unicode, including how to *encode* text strings to bytes and *decode* bytes back to strings. You'll see that the correspondence between bytes and text characters can be expressed in many ways. The hex string representation in the preceding section was just another way of specifying integers. But whenever we store or transport strings, they're reduced to bytes, and we have to track what they represent.

Convert to a Hex String

Here's how to go back from bytes to a hex string representation:

```
>>> the_bytes = b'\xff\x01abcdef'
>>> hex_string = the_bytes.hex()
>>> hex_string
'ff01616263646566'
>>> bytes.hex(the_bytes)
'ff01616263646566'
```

Get One Byte by [offset]

As with strings and other sequences (tuples and lists in Chapter 8), use square brackets and specify the index (position, starting at 0) of the byte that you want:

```
>>> b = b'\xff\x01abcdef'
>>> b[2]
97
>>> chr(b[2])
'a'
```

That `b[2]` was specified as `a`, but that really is the same as `\x61`, which is a decimal `97`. Here, the `chr()` function converts an integer value to its ASCII equivalent. (Actually, it converts an integer Unicode value, which has millions of entries. Much more on `chr()` and others in Chapter 17.)

Get a Slice

As with strings, you can get more than one element with a slice:

```
>>> b = b'\xff\x01abcdef'
>>> b[0:2]
b'\xff\x01'
>>> b[-3:]
b'def'
```

Combine with +

```
>>> b1 = bytes((1, 2, 3))
>>> b2 = bytes((4, 5, 6))
>>> b1 + b2
b'\x01\x02\x03\x04\x05\x06'
```

A `bytes` value is immutable, but you can *appear* to change it. You're actually creating a new `bytes` value, then assigning it to the old *variable* name. Remember from Chapter 2 that in Python a variable is just a *reference* (or name, or label) to a value somewhere in memory, not the value itself. Let's append the contents of `b2` to `b1`:

```
>>> b1 += b2
>>> b1
b'\x01\x02\x03\x04\x05\x06'
```

This is the same as saying `b1 = b1 + b2`.

Python takes the `bytes` values that `b1` and `b2` refer to, combines them into a brand-new `bytes` value, then assigns that value back to the old `b1` variable name. The old `b1` value is now gone forever.

Repeat with *

Repeat a sequence of bytes by using `*`:

```
>>> b = b'\xff\x01abcdef'
>>> b * 2
b'\xff\x01abcdef\xff\x01abcdef'
```

Bytearray

Now we'll delve into the mutable cousin of `bytes`: `bytearray`. The first (read-only) group of operations that follow are similar to those of immutable `bytes` and strings.

Create with bytearray()

The function `bytearray()` is, shockingly, used to create a Python `bytearray`. If you don't provide any arguments (values inside the parentheses), you'll get an empty `bytearray`:

```
>>> b = bytearray()
>>> b
bytearray(b'')
```

Provide a list (see Chapter 8) and see its elements inserted into the `bytearray`:

```
>>> blist = [1, 2, 3, 255]
>>> ba = bytearray(blist)
>>> ba
bytearray(b'\x01\x02\x03\xff')
```

Similarly, provide a tuple (also in Chapter 8):

```
>>> btuple = 4, 5, 6
>>> ba = bytearray(btuple)
>>> ba
bytearray(b'\x04\x05\x06')
```

If you give the `bytearray()` function a single integer, you'll get a `bytearray` of that length, with each element equal to zero:

```
>>> ba = bytearray(5)
>>> ba
bytearray(b'\x00\x00\x00\x00\x00')
```

Finally, if you provide a text string, you'll also need to specify the *encoding* (how the string characters map to their byte equivalents). You may learn more about encoding than you care to in Chapters 19 and 20.

```
>>> ba = bytearray("abc", "ascii")
>>> ba
bytearray(b'abc')
```

Get One Byte by [offset]

This is also called *indexing*, and it works as it does for `bytes`:

```
>>> ba = bytearray(b'\xff\x01abcdef')
>>> ba[2]
97
```

Get Multiple Bytes with a Slice

As with strings and `bytes`, you can access multiple elements with a slice:

```
>>> ba = bytearray(b'\xff\x01abcdef')
>>> ba[4:6]
bytearray(b'cd')
```

Modify One Byte by [offset]

Now we get into the mutable stuff and do things that we can't with `bytes`, like change an element in place:

```
>>> ba = bytearray(b'\x01\x02\x03\xff')
>>> ba[1] = 127
>>> ba
bytearray(b'\x01\x7f\x03\xff')
```

Modify Multiple Bytes with replace()

The `replace()` method finds the byte sequence that matches its first argument and replaces it with its second argument:

```
>>> ba = bytearray(b'\xff\x01abcdef')
>>> ba
bytearray(b'\xff\x01abcdef')
>>> ba.replace(b'\x01ab', b'XYZ\x6F')
bytearray(b'\xffXYZocdef')
```

Modify Multiple Bytes with a Slice

Now the slice can be on the lefthand side of the = assignment:

```
>>> ba = bytearray(b'\xff\x01abcdef')
>>> ba[2:5] = b'oops'
>>> ba
bytearray(b'\xff\x01oopsdef')
```

Notice that in the previous example and this one, the size and contents of the bytearray change, and that you can't do this with a text string.

Insert a Byte with insert()

Use `insert(`*`index, value`*`)` to insert the byte value at the offset index:

```
>>> ba = bytearray(b'\xff\x01abcdef')
>>> ba.insert(0, 2)
>>> ba
bytearray(b'\x02\xff\x01abcdef')
```

Append One Byte with append()

Add a byte to the end of a `bytearray`:

```
>>> ba = bytearray(b'\xff\x01abcdef')
>>> ba.append(4)
>>> ba
bytearray(b'\xff\x01abcdef\x04')
```

Unfortunately, the argument to `append()` must be an integer, not a `bytes` object:

```
>>> ba.append(b'\x04')
Traceback (most recent call last):
  File "<python-input-45>", line 1, in <module>
    ba.append(b'\x04')
    ~~~~~~~~~^^^^^^^^
TypeError: 'bytes' object cannot be interpreted as an integer
```

Append Multiple Bytes with extend()

Use `extend()` to tack bytes onto the end of a `bytearray`:

```
>>> ba = bytearray(b'\xff\x01abcdef')
>>> ba.extend(b'!!!')
```

```
>>> ba
bytearray(b'\x02\xff\x01abcdef!!!')
```

Combine with +

Combine bytearrays with +:

```
>>> ba = bytearray(b'\x01\x02') + bytearray(b'\x03\x04')
>>> ba
bytearray(b'\x01\x02\x03\x04)
```

Repeat with *

Duplicate a bytearray with *:

```
>>> ba = bytearray(b'\x01\x02') * 2
>>> ba
bytearray(b'\x01\x02\x01\x02')
```

Review/Preview

Unlike text strings, which have only an immutable version, byte sequences can be mutable (`bytearray`) or immutable (`bytes`).

In the next chapter, we switch gears a bit and move from basic *data* structures (numbers, strings, bytes) to basic *code* structures that let us make decisions and express comparisons: `if` and `match`.

Practice

5.1 Make a `bytes` variable called `b` from the integers 1 through 5.

5.2 Print the length of `b`.

5.3 Make a `bytearray` variable called `ba`. Or call it `sasquatch`; it really doesn't matter. Fill the variable with the integers 1 through 5.

5.4 Print the length of `ba` (or whatever you called it).

5.5 Assign `b` to `ba`.

5.6 Assign `ba` to `b`.

5.7 Repeat the contents of `b`.

5.8 Repeat the contents of `ba`.

CHAPTER 6

If and Match

If you can keep your head when all about you
Are losing theirs and blaming it on you, ...
—Rudyard Kipling, "If—"

In the previous chapters, you've seen many examples of data but haven't done much with them. Most of the code examples used the interactive interpreter and were short. In this chapter, you'll learn how to structure Python *code*, not just data.

Many computer languages use characters such as curly braces (`{` and `}`) or keywords such as `begin` and `end` to mark off sections of code. In those languages, it's good practice to use consistent indentation to make your program more readable for yourself and others. There are even tools to make your code line up nicely.

When he was designing the language that became Python, Guido van Rossum decided that the indentation itself was enough to define a program's structure, and avoided typing all those parentheses and curly braces. Python is unusual in this use of *whitespace* to define program structure. It's one of the first aspects that newcomers notice, and it can seem odd to those who have experience with other languages. It turns out that after writing Python for a little while, that structure feels natural, and you stop noticing it. You even get used to doing more while typing less.

Our initial code examples have been one-liners. Let's first see how to make comments and multiple-line commands.

Comment with

A *comment* is a piece of text in your program that is ignored by the Python interpreter. You might use comments to clarify nearby Python code, make notes to yourself to fix something someday, or for whatever purpose you like. You mark a

comment by using the # character; everything from that point on to the end of the current line is part of the comment. You'll usually see a comment on a line by itself, as shown here:

```
>>> # 60 sec/min * 60 min/hr * 24 hr/day
>>> seconds_per_day = 86400
```

Or on the same line as the code it's commenting:

```
>>> seconds_per_day = 86400 # 60 sec/min * 60 min/hr * 24 hr/day
```

The # character has many names: *hash*, *sharp*, *pound*, or the sinister-sounding *octothorpe*.[1] Whatever you call it,[2] its effect lasts only to the end of the line on which it appears.

Python does not have a multiline comment. You need to explicitly begin each comment line or section with a #:

```
>>> # I can say anything here, even if Python doesn't like it,
... # because I'm protected by the awesome
... # octothorpe.
...
>>>
```

However, if it's in a text string, the mighty octothorpe reverts back to its role as a plain old # character:

```
>>> print("No comment: quotes make the # harmless.")
No comment: quotes make the # harmless.
```

Continue Lines with \

Programs are more readable when lines are reasonably short. The recommended (not required) maximum line length is 80 characters. If you can't say everything you want to say in that length, you can use the *continuation character*: \ (backslash). Just put \ at the end of a line, and Python will suddenly act as though you're still on the same line.

For example, if I wanted to add the first five digits, I could do it a line at a time:

```
>>> sum = 0
>>> sum += 1
>>> sum += 2
>>> sum += 3
>>> sum += 4
>>> sum
10
```

1 Like that eight-legged green *thing* that's *right behind you*!

2 Please don't call it. It might come back.

Or I could do it in one step, using the continuation character to escape the newlines:

```
>>> sum = 1 + \
...       2 + \
...       3 + \
...       4
>>> sum
10
```

If we skipped the backslash in the middle of an expression, we'd get an exception:

```
>>> sum = 1 +
  File "<stdin>", line 1
    sum = 1 +
              ^
SyntaxError: invalid syntax
```

Here's the preferred way to make multiline expressions: Python doesn't squawk about line endings if you're in the middle of paired parentheses (or square brackets or curly braces):

```
>>> sum = (
...     1 +
...     2 +
...     3 +
...     4)
>>>
>>> sum
10
```

You also saw in Chapter 4 that paired triple quotes let you make multiline strings.

Compare with if, elif, and else

Now, we finally take our first step into the *code structures* that weave data into programs. Our first example is this tiny Python program that checks the value of the Boolean variable `disaster` and prints an appropriate comment:

```
>>> disaster = True
>>> if disaster:
...     print("Woe!")
... else:
...     print("Whee!")
...
Woe!
>>>
```

The `if` and `else` lines are Python *statements* that check whether a condition (here, the value of `disaster`) is a Boolean `True` value, or can be evaluated as `True`. Remember, `print()` is Python's built-in *function* to print things, normally to your screen.

If you've programmed in other languages, note that you don't need parentheses for the `if` test. For example, `if disaster` is preferred to `if (disaster)` or `if (disaster == True)` (the equality operator `==` is described in a few paragraphs). You do need the colon (`:`) at the end. If, like me, you forget to type the colon at times, Python will display an error message.

Each `print()` line is indented under its test. I used four spaces to indent each subsection. Although you can use any indentation you like, Python expects you to be consistent with code within a section—the lines need to be indented the same amount, lined up on the left. The recommended style, called *PEP-8*, is to use four spaces. Don't mix tabs and spaces; it messes up the indent count.[3]

We took these steps here, which I explain more fully as the chapter progresses:

- Assigned the Boolean value `True` to the variable named `disaster`
- Performed a *conditional comparison* by using `if` and `else`, executing different code depending on the value of `disaster`
- *Called* the `print()` *function* to print some text

You can have tests within tests, as many levels deep as needed:

```
>>> furry = True
>>> large = True
>>> if furry:
...     if large:
...         print("It's a yeti.")
...     else:
...         print("It's a cat!")
... else:
...     if large:
...         print("It's a whale!")
...     else:
...         print("It's a human. Or a hairless cat.")
...
It's a yeti.
```

In Python, indentation determines how the `if` and `else` sections are paired. Our first test was to check `furry`. Because `furry` is `True`, Python goes to the indented `if large` test. Because we had set `large` to `True`, `if large` is evaluated as `True`, and the following `else` line is ignored. This makes Python run the line indented under `if large:` and print `It's a yeti.`

3 The HBO series *Silicon Valley* devoted a whole episode to tabs versus spaces.

If there are more than two possibilities to test, use `if` for the first, `elif` (meaning *else if*) for the middle ones, and `else` for the last:

```
>>> color = "mauve"
>>> if color == "red":
...     print("It's a tomato")
... elif color == "green":
...     print("It's a green pepper")
... elif color == "bee purple":
...     print("I don't know what it is, but only bees can see it")
... else:
...     print("I've never heard of the color", color)
...
I've never heard of the color mauve
```

(If you're new to programming, the preceding lines are executed one after another, in the order they appear. So the `else` is run only after all the other checks were run and were evaluated to be false.)

In the preceding example, we tested for equality by using the `==` operator. Table 6-1 shows Python's *comparison operators*.

Table 6-1. Comparison operators

Usage	Operator
Equality	`==`
Inequality	`!=`
Less than	`<`
Less than or equal	`<=`
Greater than	`>`
Greater than or equal	`>=`

These return the Boolean values `True` or `False`. Let's see how these all work, but first, assign a value to `x`:

```
>>> x = 7
```

Now, let's try some tests:

```
>>> x == 5
False
>>> x == 7
True
>>> 5 < x
True
>>> x < 10
True
```

Note that two equals signs (==) are used to *test equality*; remember, a single equals sign (=) is what you use to assign a value to a variable.

If you need to make multiple comparisons at the same time, you use the *logical* (or *Boolean*) *operators* `and`, `or`, and `not` to determine the final Boolean result.

Logical operators have lower *precedence* than the chunks of code that they're comparing. This means that the chunks are calculated first and then compared. In this example, because we set `x` to `7`, `5 < x` is calculated to be `True` and `x < 10` is also `True`, so we finally end up with `True and True`:

```
>>> 5 < x and x < 10
True
```

The easiest way to avoid confusion about precedence is to add parentheses:

```
>>> (5 < x) and (x < 10)
True
```

Here are some other tests:

```
>>> 5 < x or x < 10
True
>>> 5 < x and x > 10
False
>>> 5 < x and not x > 10
True
```

If you're `and`-ing multiple comparisons with one variable, Python lets you do this:

```
>>> 5 < x < 10
True
```

It's the same as `5 < x and x < 10`. You can also write longer comparisons:

```
>>> 5 < x < 10 < 999
True
```

What Is True?

What if the element we're checking isn't a Boolean? What does Python consider `True` and `False`?

A `false` value doesn't necessarily need to explicitly be a Boolean `False`. For example, Table 6-2 lists values that are all considered `False`.

Table 6-2. False equivalents

Interpretation	Literal code
Boolean	`False`
Null	`None`
Zero integer	`0`
Zero float	`0.0`
Empty string	`''`
Empty list	`[]`
Empty tuple	`()`
Empty dict	`{}`
Empty set	`set()`

Anything else is considered `True`. Python programs use these definitions of "truthiness" and "falsiness" to check for empty data structures as well as `False` conditions:

```
>>> some_list = []
>>> if some_list:
...     print("There's something in here")
... else:
...     print("Hey, it's empty!")
...
Hey, it's empty!
```

If what you're testing is an expression rather than a simple variable, Python evaluates the expression and returns a Boolean result. Say you type this:

```
if color == "red":
```

Python evaluates `color == "red"`. In our earlier example, we assigned the string `mauve` to `color`, so `color == "red"` is `False`, and Python moves on to the next test:

```
elif color == "green":
```

Do Multiple Comparisons with in

Suppose that you have a letter and want to know whether it's a vowel. One way would be to write a long `if` statement:

```
>>> letter = 'o'
>>> if letter == 'a' or letter == 'e' or letter == 'i' \
...     or letter == 'o' or letter == 'u':
...     print(letter, 'is a vowel')
... else:
...     print(letter, 'is not a vowel')
...
o is a vowel
>>>
```

Whenever you need to make a lot of comparisons like that, separated by `or`, use Python's *membership operator* `in`, instead. Here's how to check vowel-ness more Pythonically, using `in` with a string made of vowel characters:

```
>>> vowels = 'aeiou'
>>> letter = 'o'
>>> letter in vowels
True
>>> if letter in vowels:
...     print(letter, 'is a vowel')
...
o is a vowel
```

Here's a preview of how to use `in` with some data types that you'll read about in detail in the next few chapters:

```
>>> letter = 'o'
>>> vowel_set = {'a', 'e', 'i', 'o', 'u'}
>>> letter in vowel_set
True
>>> vowel_list = ['a', 'e', 'i', 'o', 'u']
>>> letter in vowel_list
True
>>> vowel_tuple = ('a', 'e', 'i', 'o', 'u')
>>> letter in vowel_tuple
True
>>> vowel_dict = {'a': 'apple', 'e': 'elephant',
...               'i': 'impala', 'o': 'ocelot', 'u': 'unicorn'}
>>> letter in vowel_dict
True
>>> vowel_string = "aeiou"
>>> letter in vowel_string
True
```

For the dictionary, `in` looks at the keys (the lefthand side of the `:`) instead of their values.

New: I Am the Walrus

Arriving in Python 3.8 is the *walrus operator*, which looks like this:

```
name := expression
```

See the walrus? (Like a smiley, but tuskier.)

Normally, an assignment and test take two steps:

```
>>> post_limit = 300
>>> post_string = "Blah" * 50
>>> diff = post_limit - len(post_string)
>>> if diff >= 0:
...     print("A fitting post")
```

```
... else:
...     print("Went over by", abs(diff))
...
A fitting post
```

With our new tuskiness (aka assignment expressions), we can combine these into one step:

```
>>> post_limit = 300
>>> post_string = "Blah" * 50
>>> if (diff := post_limit - len(post_string)) >= 0:
...     print("A fitting post")
... else:
...     print("Went over by", abs(diff))
...
A fitting post
```

We needed those parentheses around the comparison in the preceding example. Without them, because of precedence rules, `if diff := post_limit - len(post_string) >= 0` would have been interpreted as `if (diff := ((post_limit - len(post_string)) >= 0))`, which just evaluates to `True` and would have accidentally succeeded here.

The walrus also gets on swimmingly with `for` and `while`, which we look at in Chapter 7.

Match

The `match` statement is a recent (version 3.10) addition to Python. It's similar to the `switch` statement in other languages like C and Java. You give it a *subject*, then one or more *patterns* to match against that subject's value and/or type:

```
match subject:
    case pattern1:
        # ...
    case pattern2:
        # ...
    # other patterns ...
    case _:
        # if nothing else matches
```

You can do the same job with a bunch of `if`, `else`, and `elif` statements. So you may not really need to use `match`. It's sometimes a bit more compact—and, by some benchmarks, a bit faster.

This is called *structural* pattern matching, to distinguish it from pure text matching. You'll see the difference in the coming examples.

This `match` is not the same as the `match()` function in the regular expression library that you'll see in Chapter 17.

Simple Matches

The simplest use of `match` is like the `switch` statement in languages like C—except you don't use something like `break` to avoid "falling through" to the next case. If a case does match, its code is executed, and the `match` statement finishes:

```
>>> superhero = "Spiderman"
... match superhero:
...     case "Superman":
...         secret_identity = "Clark Kent"
...     case "Batman":
...         secret_identity = "Bruce Wayne"
...     case "Spiderman":
...         secret_identity = "Peter Parker"
...     case _:
...         secret_identity = "?"
...
... print(secret_identity)
Peter Parker
```

You could have used `if ... elif ... else` instead:

```
>>> superhero = "Spiderman"
... if superhero == "Superman":
...     secret_identity = "Clark Kent"
... elif superhero == "Batman":
...     secret_identity = "Bruce Wayne"
... elif superhero == "Spiderman":
...     secret_identity = "Peter Parker"
... else:
...     secret_identity = "?"
... print(secret_identity)
Peter Parker
```

Or, more succinctly (a sneak peek at dictionaries in Chapter 9):

```
>>> superhero = "Spiderman"
... secret_identities = {
...     "Superman": "Clark Kent",
...     "Batman": "Bruce Wayne",
...     "Spiderman": "Peter Parker"
... }
... secret_identity = secret_identities.get(superhero, "?")
>>> print(secret_identity)
Peter Parker
```

That second argument (`?`) to `secret_identities.get()` serves the same purpose as the `case _` in the first example and the `else` in the second example: catch any mismatch.

Structural Matches

In this example, the *subject* contains multiple values, and we want to match *partly* on a given pattern.

Even though the subject is a list, it doesn't matter if you express the patterns as lists or tuples. They're both sequences:

```
>>> subject = [3, 5]
... match subject:
...     case x, y if y > 6:
...         print("y > 6")
...     case 2, 5:
...         print("2, 5")
...     case x, 5:
...         print(f"{x=}", f"{y=}")
...     case 3, 5:
...         print("3, 5")
...     case _:
...         print("no match for", subject)
x=3 y=5
```

Note the following:

- The first pattern (`x, y`) contains the *guard* `if y > 6`. And that isn't true, so …
- The second pattern (`[2, 5]`) doesn't match the first value (`3`) in the subject, so we carry on.
- Aha! The third pattern (`[x, 5]`) matches, so we're done.
- Which is too bad for the fourth pattern, which would have been an exact match. If this had been the third pattern instead, it would have won.
- And no luck for the default `_`.

Review/Preview

You can make decisions with `if`, `elif`, and `else`. These are the first Python control structures. If you're comparing a variable with more than one pattern, `match` is an alternative to `if` … `elif` … `else` lines.

The next chapter introduces `for` and `while`, which let you express repetitions (loops).

Practice

6.1 Choose a number from 1 to 10 and assign it to the variable `secret`. Then select another number from 1 to 10 and assign it to the variable `guess`. Next, write the conditional tests (`if`, `else`, and `elif`) to print the string `too low` if `guess` is less than `secret`, or `too high` if greater than `secret`, or `just right` if equal to `secret`.

6.2 Assign `True` or `False` to the variables `small` and `green`. Write `if`/`else` statements to print which of these matches those choices: cherry, pea, watermelon, pumpkin.

CHAPTER 7

For and While

For a' that, an' a' that,
Our toils obscure, an' a' that ...

—Robert Burns, "For a' That and a' That"

Testing with `if`, `elif`, and `else` runs down the page, a line at a time, from top to bottom. Sometimes we need to do something more than once. We need a *loop*, and Python gives us two choices: `while` and `for`.

Repeat with while

The simplest looping mechanism in Python is `while`. Using the interactive interpreter, try a simple loop that prints the numbers from 1 to 5:

```
>>> count = 1
>>> while count <= 5:
...     print(count)
...     count += 1
...
1
2
3
4
5
>>>
```

We first assign the value `1` to `count`. The `while` loop compares the value of `count` to `5` and continues if `count` is less than or equal to `5`. Inside the loop, we print the value of `count` and then *increment* its value by one with the statement `count += 1`. Python goes back to the top of the loop and again compares `count` with `5`. The value of `count` is now `2`, so the contents of the `while` loop are again executed, and `count` is incremented to `3`.

This continues until `count` is incremented from `5` to `6` at the bottom of the loop (the `count += 1` line). On the next trip to the top, `count` is now `6`, so `count <= 5` is now `False`, and the `while` loop ends. Python moves on to the next lines.

Cancel with break

If you want to loop until something occurs, but you're not sure when that might happen, you can use an *infinite loop* with a `break` statement. This time, let's read a line of input from the keyboard via Python's `input()` function and then print it with the first letter capitalized. We break out of the loop when a line containing only the letter `q` is typed:

```
>>> while True:
...     stuff = input("String to capitalize [type q to quit]: ")
...     if stuff == "q":
...         break
...     print(stuff.capitalize())
...
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
>>>
```

Skip Ahead with continue

Sometimes you don't want to break out of a loop but just want to skip ahead to the next iteration for some reason. Here's a contrived example: let's read an integer, print its square if it's odd, and skip it if it's even. I even added a few comments. Again, we use `q` to stop the loop:

```
>>> while True:
...     value = input("Integer, please [q to quit]: ")
...     if value == 'q':      # quit
...         break
...     number = int(value)
...     if number % 2 == 0:   # an even number
...        continue
...     print(number, "squared is", number*number)
...
Integer, please [q to quit]: 1
1 squared is 1
Integer, please [q to quit]: 2
Integer, please [q to quit]: 3
3 squared is 9
Integer, please [q to quit]: 4
Integer, please [q to quit]: 5
5 squared is 25
```

```
Integer, please [q to quit]: q
>>>
```

Check break Use with else

If the `while` loop ends normally (no `break` call), control passes to an optional `else`. You use this when you've coded a `while` loop to check for something and to break as soon as it's found. The `else` runs if the `while` loop completes but the object is not found:

```
>>> numbers = [1, 3, 5]
>>> position = 0
>>> while position < len(numbers):
...     number = numbers[position]
...     if number % 2 == 0:
...         print('Found even number', number)
...         break
...     position += 1
... else:  # break not called
...     print('No even number found')
...
No even number found
```

This use of `else` might seem nonintuitive. Consider it a *break checker*. Notice that the `else` is left-aligned with the `while`, not the `if`.

Iterate with for and in

Python makes frequent use of *iterators*, for good reason. They make it possible for you to traverse data structures without knowing how large they are or how they are implemented. You can even iterate over data that is created on the fly, allowing processing of data streams that would otherwise not fit in the computer's memory all at once.

To show iteration, we need something to iterate over. You've already seen strings in Chapter 4 but have not yet read the details on other *iterables* like lists and tuples (Chapter 8) or dictionaries (Chapter 9). I'll show two ways to walk through a string here, and show iteration for the other types in their own chapters.

It's legal Python to step through a string like this:

```
>>> word = 'thud'
>>> offset = 0
>>> while offset < len(word):
...     print(word[offset])
...     offset += 1
```

```
...
t
h
u
d
```

But there's a better way in Python:

```
>>> for letter in word:
...     print(letter)
...
t
h
u
d
```

String iteration produces one character at a time.

Cancel with break

A `break` in a `for` loop breaks out of the loop, as it does for a `while` loop:

```
>>> word = 'thud'
>>> for letter in word:
...     if letter == 'u':
...         break
...     print(letter)
...
t
h
```

Skip with continue

Inserting a `continue` in a `for` loop jumps to the next iteration of the loop, as it does for a `while` loop.

Check break Use with else

Similar to `while`, `for` has an optional `else` that checks whether the `for` has completed normally. If `break` was *not* called, the `else` statement is run.

This is useful when you want to verify that the previous `for` loop ran to completion instead of being stopped early with a `break`:

```
>>> word = 'thud'
>>> for letter in word:
...     if letter == 'x':
...         print("Eek! An 'x'!")
...         break
...     print(letter)
... else:
...     print("No 'x' in there.")
```

```
...
t
h
u
d
No 'x' in there.
```

As with `while`, the use of `else` with `for` might seem nonintuitive. It makes more sense if you think of the `for` as looking for something, and `else` being called if you didn't find it. To get the same effect without `else`, use a variable to indicate whether you found what you wanted in the `for` loop.

Generate Number Sequences with range()

The `range()` function returns an object that generates a stream of numbers within a specified range, without first having to create and store a large data structure such as a list or tuple. This lets you create huge ranges without using all the memory in your computer and crashing your program.

You use `range()` similarly to the way you use slices: `range(` *`start`*, *`stop`*, *`step`* `)`. If you omit *`start`*, the range begins at `0`. The only required value is *`stop`*; as with slices, the last value created will be just before *`stop`*. The default value of *`step`* is `1`, but you can go backward with `-1`.

The object returned by `range()` is *iterable*, so you need to step through the values with `for ... in`, or convert the object to a sequence like a list. Let's make the range `0, 1, 2`:

```
>>> for x in range(0,3):
...     print(x)
...
0
1
2
>>> list( range(0, 3) )
[0, 1, 2]
```

Here's how to make a range from `2` down to `0`:

```
>>> for x in range(2, -1, -1):
...     print(x)
...
2
1
0
>>> list( range(2, -1, -1) )
[2, 1, 0]
```

The following snippet uses a step size of 2 to get the even numbers from 0 to 10:

```
>>> list( range(0, 11, 2) )
[0, 2, 4, 6, 8, 10]
```

Other Iterators

Chapter 22 shows iteration over files. In Chapter 11, you can see how to enable iteration over *objects* that you've defined yourself. Also, Chapter 12 talks about `itertools`, a standard Python module with many useful shortcuts.

Review/Preview

We progressed from line-at-a-time execution to loops, using `for` and `while`. Using `for` with `in` is an example of iteration, which is a core Python pattern.

In the next chapter, we go back to data types: tuples and lists, which can contain any number of any type of data.

Practice

7.1 Use a `for` loop to print the values of the list `[3, 2, 1, 0]`.

7.2 Assign the value `7` to the variable `guess_me`, and the value `1` to the variable `number`. Write a `while` loop that compares `number` with `guess_me`. Print `too low` if `number` is less than `guess me`. If `number` equals `guess_me`, print `found it!` and then exit the loop. If `number` is greater than `guess_me`, print `oops` and then exit the loop. Increment `number` at the end of the loop.

7.3 Assign the value `5` to the variable `guess_me`. Use a `for` loop to iterate a variable called `number` over `range(10)`. If `number` is less than `guess_me`, print `too low`. If it equals `guess_me`, print `found it!` and then break out of the `for` loop. If `number` is greater than `guess_me`, print `oops` and then exit the loop.

CHAPTER 8
Tuples and Lists

The human animal differs from the lesser primates in his passion for lists.
—H. Allen Smith

In the previous chapters, we started with some of Python's basic data types: Booleans, integers, floats, and strings. If you think of those as atoms, the data structures in this chapter are like molecules. That is, we combine those basic types in more complex ways. You will use these every day. Much of programming consists of chopping and gluing data into specific forms, and the tuples and lists of this chapter are some of our first examples.

Most computer languages can represent a sequence of items indexed by their integer position: first, second, and so on down to the last. You've already seen Python strings (Chapter 4; sequences of text characters) and bytes and bytearrays (Chapter 5; sequences of 8-bit binary values).

Python has more sequence structures: tuples and lists. These contain zero or more elements. Unlike strings, the elements can be of different types. In fact, each element can be *any* Python object. This lets you create structures as deep and complex as you like.

Why does Python contain both lists and tuples? Tuples are *immutable*; when you assign elements (only once) to a tuple, they're baked in the cake and can't be changed. Lists are *mutable*, meaning you can insert and delete elements with great enthusiasm. I'll show many examples of each, with an emphasis on lists.

Tuples

Let's get one detail out of the way first. You may hear two pronunciations for *tuple*. Which is right? If you guess wrong, do you risk being considered a Python poseur? No worries. Guido van Rossum, the creator of Python, said via Twitter:

> I pronounce tuple too-pull on Mon/Wed/Fri and tub-pull on Tue/Thu/Sat. On Sunday I don't talk about them. :)

The original Latin word *tuple*, meaning *plus*, was pronounced *tub-pull*.

Create with Commas and ()

The syntax to make tuples is a little inconsistent, as the following examples demonstrate. Let's begin by making an empty tuple using `()`:

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

To make a tuple with one or more elements, follow each element with a comma. This works for one-element tuples:

```
>>> one_marx = 'Groucho',
>>> one_marx
('Groucho',)
```

You could enclose them in parentheses and still get the same tuple:

```
>>> one_marx = ('Groucho',)
>>> one_marx
('Groucho',)
```

Here's a little gotcha: if you have a single thing in parentheses and omit that comma, you would not get a tuple, but just the thing (in this example, the string `Groucho`):

```
>>> one_marx = ('Groucho')
>>> one_marx
'Groucho'
>>> type(one_marx)
<class 'str'>
```

If you have more than one element, follow all but the last one with a comma:

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Actually, having a comma after the last element doesn't hurt (some languages are more picky about this):

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo',
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Python includes parentheses when echoing a tuple. You often don't need them when you define a tuple, but using parentheses is a little safer, and it helps to make the tuple more visible:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

You do need the parentheses when commas might also have another use. In this example, you can create and assign a single-element tuple with just a trailing comma, but you can't pass it as an argument to a function:

```
>>> one_marx = 'Groucho',
>>> type(one_marx)
<class 'tuple'>
>>> type('Groucho',)
<class 'str'>
>>> type(('Groucho',))
<class 'tuple'>
```

Tuples let you assign multiple variables at once:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> a, b, c = marx_tuple
>>> a
'Groucho'
>>> b
'Chico'
>>> c
'Harpo'
```

This is sometimes called *tuple unpacking*. If the number of variables doesn't match the length of the tuple, then oops:

```
>>> three_numbers = (1, 2, 3)
>>> one, two, three, four = three_numbers
Traceback (most recent call last):
  File "<python-input-8>", line 1, in <module>
    one, two, three, four = three_numbers
    ^^^^^^^^^^^^^^^^^^^^^
ValueError: not enough values to unpack (expected 4, got 3)
```

You can use tuples to exchange values in one statement without using a temporary variable:

```
>>> password = 'swordfish'
>>> icecream = 'tuttifrutti'
>>> password, icecream = icecream, password
>>> password
'tuttifrutti'
```

```
>>> icecream
'swordfish'
>>>
```

Create with tuple()

The `tuple()` conversion function makes tuples from other iterable objects:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple(b'\x01\x02\x03')
(1, 2, 3)
```

Get an Item by [offset]

As with any sequence, you get a single item from a tuple by providing its offset:

```
>>> t = 1, 2, 3
>>> t[0]
1
```

But you can't modify an item by its offset. "It's immutable!" shout the tuple's lawyers:

```
>>> t[0] = 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Combine with +

As with strings, + combines two or more tuples:

```
>>> ('Groucho',) + ('Chico', 'Harpo')
('Groucho', 'Chico', 'Harpo')
```

Duplicate with *

Use * to make copies of a tuple, and combine them:

```
>>> ('yada',) * 3
('yada', 'yada', 'yada')
```

Compare

Comparing tuples works much like list comparisons:

```
>>> a = (7, 2)
>>> b = (7, 2, 9)
>>> a == b
False
```

```
>>> a <= b
True
>>> a < b
True
```

Iterate with for and in

Tuple iteration is like iteration of other types:

```
>>> words = ('fresh','out', 'of', 'ideas')
>>> for word in words:
...     print(word)
...
fresh
out
of
ideas
```

Modify?

You can't! Like strings, tuples are immutable, so you can't change an existing one. As you saw just before, you can *concatenate* (combine) tuples to make a new one, as you can with strings:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop,')
>>> t1 + t2
('Fee', 'Fie', 'Foe', 'Flop')
```

This means that you can appear to modify a tuple like this:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop,')
>>> t1 += t2
>>> t1
('Fee', 'Fie', 'Foe', 'Flop')
```

But it isn't the same `t1`! Python made a new tuple from the original tuples pointed to by `t1` and `t2`, and assigned the name `t1` to this new tuple. In Python, variables are just names, not pointers to values. You can see with `id()` when a variable name is pointing to a new value:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> id(t1)
4365405712
>>> t1 += t2
>>> id(t1)
4364770744
```

Named Tuples

A *named tuple* is a variety of tuple that lets you access its elements by names as well as positions. I'll discuss such creatures in Chapter 11.

Lists

Lists are good for keeping track of items by their order, especially when the order and contents might change. Unlike strings, lists are mutable. You can change a list in place, add new elements, and delete or replace existing elements. The same value can occur more than once in a list.

Create with []

A list is made from zero or more elements, separated by commas and surrounded by square brackets:

```
>>> empty_list = [ ]
>>> weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> big_birds = ['emu', 'ostrich', 'cassowary']
>>> first_names = ['Graham', 'John', 'Terry', 'Terry', 'Michael']
>>> leap_years = [2000, 2004, 2008]
>>> randomness = ["Punxsatawney", {"groundhog": "Phil"}, "Feb. 2"}]
```

The `first_names` list shows that values do not need to be unique.

If you want to keep track of only unique values and don't care about order, a Python *set* might be a better choice than a list. In the previous example, `big_birds` could have been a set. We explore sets in Chapter 9.

Create or Convert with list()

You can also make an empty list with the `list()` function:

```
>>> another_empty_list = list()
>>> another_empty_list
[]
```

Python's `list()` function also converts other *iterable* data types (such as tuples, strings, sets, and dictionaries) to lists. The following example converts a string to a list of one-character strings:

```
>>> list('cat')
['c', 'a', 't']
```

This example converts a tuple to a list:

```
>>> a_tuple = ('ready', 'fire', 'aim')
>>> list(a_tuple)
['ready', 'fire', 'aim']
```

Create from a String with split()

As I mentioned in "Split with split()" on page 59, use `split()` to chop a string into a list by a separator:

```
>>> talk_like_a_pirate_day = '9/19/2024'
>>> talk_like_a_pirate_day.split('/')
['9', '19', '2024']
```

What if you have more than one separator string in a row in your original string? Well, you get an empty string as a list item:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

If you had used the two-character separator string `//` instead, you would get this:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('//')
>>>
['a/b', 'c/d', '/e']
```

Get an Item by [offset]

As with strings, you can extract a single value from a list by specifying its offset:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0]
'Groucho'
>>> marxes[1]
'Chico'
>>> marxes[2]
'Harpo'
```

Again, as with strings, negative indices count backward from the end:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

The offset has to be a valid one for this list—a position you have assigned a value previously. If you specify an offset before the beginning or after the end, you'll get an exception (error). Here's what happens if we try to get the sixth Marx brother (offset `5` counting from `0`), or the fifth before the end:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> marxes[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Get Items with a Slice

You can extract a subsequence of a list by using a *slice*:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0:2]
['Groucho', 'Chico']
```

A slice of a list is also a list.

As with strings, slices can step by values other than 1. The next example starts at the beginning and goes right by 2:

```
>>> marxes[::2]
['Groucho', 'Harpo']
```

Here, we start at the end and go left by 2:

```
>>> marxes[::-2]
['Harpo', 'Groucho']
```

And finally, here's a trick to reverse a list:

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

None of these slices change the `marxes` list itself, because we don't assign them to `marxes`. To reverse a list in place, use *`list`*`.reverse()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.reverse()
>>> marxes
['Harpo', 'Chico', 'Groucho']
```

The `reverse()` function changes the list but doesn't return its value.

As you saw with strings, a slice can specify an invalid index but will not cause an exception. It will "snap" to the closest valid index or return nothing:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[4:]
[]
>>> marxes[-6:]
['Groucho', 'Chico', 'Harpo']
>>> marxes[-6:-2]
['Groucho']
>>> marxes[-6:-4]
[]
```

Add an Item to the End with append()

The traditional way of adding items to a list is to `append()` them one by one to the end. In the previous examples, we forgot Zeppo, but that's all right because the list is mutable, so we can add him now:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

Add an Item by Offset with insert()

The `append()` function adds items only to the end of the list. When you want to add an item before any offset in the list, use `insert()`. Offset `0` inserts at the beginning. An offset beyond the end of the list inserts at the end, like `append()`, so you don't need to worry about Python throwing an exception:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.insert(2, 'Gummo')
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Harpo']
>>> marxes.insert(10, 'Zeppo')
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Harpo', 'Zeppo']
```

Duplicate with *

In Chapter 4, you saw that you can duplicate a string's characters with `*`. The same works for a list:

```
>>> ["blah"] * 3
['blah', 'blah', 'blah']
```

Combine with extend() or +

You can merge one list into another by using `extend()`. Suppose that a well-meaning person gives us a new list of Marxes called `others`, and we want to merge them into the main `marxes` list:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.extend(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Alternatively, we can use `+` or `+=`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes += others
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

If we had used `append()`, `others` would have been added as a *single* list item rather than merging its items:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.append(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', ['Gummo', 'Karl']]
```

This again demonstrates that a list can contain elements of different types. In this case, four strings and a list of two strings.

Change an Item with [offset]

Just as you can get the value of a list item by its offset, you can change it:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[2] = 'Wanda'
>>> marxes
['Groucho', 'Chico', 'Wanda']
```

Again, the list offset needs to be a valid one for this list.

You can't change a character in a string in this way, because strings are immutable. Lists are mutable. You can change the number of items a list contains as well as the items themselves.

Change Items with a Slice

The previous section showed how to get a sublist with a slice. You can also assign values to a sublist with a slice:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [8, 9]
>>> numbers
[1, 8, 9, 4]
```

The righthand thing that you're assigning to the list doesn't even need to have the same number of elements as the slice on the left:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [7, 8, 9]
>>> numbers
[1, 7, 8, 9, 4]
```

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = []
>>> numbers
[1, 4]
```

Actually, the righthand thing doesn't even need to be a list. Any Python iterable will do, separating its items and assigning them to list elements:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = (98, 99, 100)
>>> numbers
[1, 98, 99, 100, 4]
```

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = 'wat?'
>>> numbers
[1, 'w', 'a', 't', '?', 4]
```

Delete an Item by Offset with del

Our fact checkers have just informed us that Gummo was indeed one of the Marx Brothers, but Karl wasn't, and that whoever inserted him earlier was very rude. Let's fix that:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Karl']
>>> marxes[-1]
'Karl'
>>> del marxes[-1]
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo']
```

When you delete an item by its position in the list, the items that follow it move back to take the deleted item's space, and the list's length decreases by one. If we delete `Chico` from the last version of the `marxes` list, we get this as a result:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo']
>>> del marxes[1]
>>> marxes
['Groucho', 'Harpo', 'Gummo']
```

`del` is a Python *statement*, not a list method—you don't say `marxes[-1].del()`. It's sort of the reverse of assignment (`=`): it detaches a name from a Python object and can free up the object's memory if that name was the last reference to it.

Delete an Item by Value with remove()

If you're not sure or don't care about the location of the item in the list, use `remove()` to delete it by value. Goodbye, Groucho:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.remove('Groucho')
>>> marxes
['Chico', 'Harpo']
```

If you had duplicate list items with the same value, `remove()` deletes only the first one it finds.

Get an Item by Offset and Delete It with pop()

You can get an item from a list *and* delete it from the list at the same time by using `pop()`. Calling `pop()` with an offset will return the item at that offset; with no argument, `pop()` uses `-1`. So, `pop(0)` returns the head (start) of the list, and `pop()` or `pop(-1)` returns the tail (end), as shown here:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.pop()
'Zeppo'
>>> marxes
['Groucho', 'Chico', 'Harpo']
>>> marxes.pop(1)
'Chico'
>>> marxes
['Groucho', 'Harpo']
```

It's computing jargon time! Don't worry, these won't be on the final exam. If you use `append()` to add new items to the end and `pop()` to remove them from the same end, you've implemented a data structure known as a last in, first out (*LIFO*) queue. This is more commonly known as a *stack*. Calling the `pop(0)` function would create a first in, first out (*FIFO*) queue. These are useful when you want to process data with the oldest first (FIFO) or the newest first (LIFO).

Delete All Items with clear()

Python 3.3 introduced a method to clear a list of all its elements:

```
>>> work_quotes = ['Working hard?', 'Quick question!', 'Number one ...']
>>> work_quotes
['Working hard?', 'Quick question!', 'Number one priorities!']
>>> work_quotes.clear()
>>> work_quotes
[]
```

Find an Item's Offset by Value with index()

If you want to know the offset of an item in a list by its value, use `index()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.index('Chico')
1
```

If the value is in the list more than once, only the offset of the first one is returned:

```
>>> simpsons = ['Lisa', 'Bart', 'Marge', 'Homer', 'Bart']
>>> simpsons.index('Bart')
1
```

Test for a Value with in

The Pythonic way to check for the existence of a value in a list is to use `in`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> 'Groucho' in marxes
True
>>> 'Bob' in marxes
False
```

The same value may be in more than one position in the list. As long as it's in there at least once, `in` will return `True`:

```
>>> words = ['a', 'deer', 'a' 'female', 'deer']
>>> 'deer' in words
True
```

If you often check for the existence of a value in a list and don't care about the order of items, a Python *set* is a more appropriate way to store and look up unique values. Read about sets in Chapter 9.

Count Occurrences of a Value with count()

To count the number of times a particular value occurs in a list, use `count()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.count('Harpo')
1
>>> marxes.count('Bob')
0

>>> snl_skit = ['cheeseburger', 'cheeseburger', 'cheeseburger']
>>> snl_skit.count('cheeseburger')
3
```

Convert a List to a String with join()

"Combine with join()" on page 59 discussed `join()` in greater detail, but here's another example of what you can do with it:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> ', '.join(marxes)
'Groucho, Chico, Harpo'
```

You might be thinking that this seems a little backward. The function `join()` applies to a string, not a list. You can't say `marxes.join(', ')`, even though it seems more intuitive. The argument to `join()` is a string or any iterable sequence of strings (including a list), and its output is a string. If `join()` were just a list method, you couldn't use it with other iterable objects such as tuples or strings. If you did want it to work with any iterable type, you'd need special code for each type to handle the actual joining. It might help to remember: `join()` *is the opposite of* `split()`, as shown here:

```
>>> friends = ['Harry', 'Hermione', 'Ron']
>>> separator = ' * '
>>> joined = separator.join(friends)
>>> joined
'Harry * Hermione * Ron'
>>> separated = joined.split(separator)
>>> separated
['Harry', 'Hermione', 'Ron']
>>> separated == friends
True
```

Reorder Items with sort() or sorted()

You'll often need to sort the items in a list by their values rather than their offsets. Python provides two functions:

- The list method `sort()` sorts the list itself, *in place*. It returns `None`.
- The general function `sorted()` returns a sorted *copy* of the list.

If the items in the list are numeric, they're sorted by default in ascending numeric order. If they're strings, they're sorted in alphabetical order:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> sorted_marxes = sorted(marxes)
>>> sorted_marxes
['Chico', 'Groucho', 'Harpo']
```

Now, `sorted_marxes` is a new list, and creating it did not change the original list:

```
>>> marxes
['Groucho', 'Chico', 'Harpo']
```

But calling the list function `sort()` on the `marxes` list does change `marxes`:

```
>>> marxes.sort()
>>> marxes
['Chico', 'Groucho', 'Harpo']
```

If the elements of your list are all of the same type (such as strings in `marxes`), `sort()` will work correctly. You can sometimes even mix types—for example, integers and floats—because they are automatically converted to one another by Python in expressions:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4.0]
```

The default sort order is ascending, but you can add the argument `reverse=True` to set the order to descending:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort(reverse=True)
>>> numbers
[4.0, 3, 2, 1]
```

Get Length with len()

The `len()` function returns the number of items in a list:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> len(marxes)
3
```

Assign with =

When you assign one list to more than one variable, changing the list in one place also changes it in the other, as illustrated here:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = 'surprise'
>>> a
['surprise', 2, 3]
```

So what's in `b` now? Is it still `[1, 2, 3]`, or `['surprise', 2, 3]`? Let's see:

```
>>> b
['surprise', 2, 3]
```

Remember the box (object) and string with note (variable name) analogy in Chapter 2? The variable `b` just refers to the same list object as `a` (both name strings lead to the same object box). Whether we change the list contents by using the name `a` or `b`, it's reflected in both:

```
>>> b
['surprise', 2, 3]
>>> b[0] = 'I hate surprises'
>>> b
['I hate surprises', 2, 3]
>>> a
['I hate surprises', 2, 3]
```

All mutable types in Python share this behavior.

Copy with copy(), list(), or a Slice

You can *copy* the values of a list to an independent, fresh list by using any of these methods:

- The list `copy()` method
- The `list()` conversion function
- The list slice `[:]`

Our original list will be called `a` again. We make `b` from `a` with the list `copy()` function, `c` with the `list()` conversion function, and `d` with a list slice:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
```

```
>>> c = list(a)
>>> d = a[:]
```

Again, `b`, `c`, and `d` are *copies* of `a`: they are new objects with their own values and no connection to the original list object `[1, 2, 3]` to which `a` refers. Changing `a` does *not* affect the copies `b`, `c`, and `d`:

```
>>> a[0] = 'integer lists are boring'
>>> a
['integer lists are boring', 2, 3]
>>> b
[1, 2, 3]
>>> c
[1, 2, 3]
>>> d
[1, 2, 3]
```

Copy Everything with deepcopy()

The `copy()` function works well if the list values are all immutable. As you've seen before, mutable values (like lists, sets, or dicts) are references. A change in the original or the copy would be reflected in both.

Let's use the previous example but make the last element in list `a` the list `[8, 9]` instead of the integer `3`:

```
>>> a = [1, 2, [8, 9]]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
>>> a
[1, 2, [8, 9]]
>>> b
[1, 2, [8, 9]]
>>> c
[1, 2, [8, 9]]
>>> d
[1, 2, [8, 9]]
```

So far, so good. Now change an element in that sublist in `a`:

```
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 10]]
>>> c
[1, 2, [8, 10]]
>>> d
[1, 2, [8, 10]]
```

The value of `a[2]` is now a list, and its elements can be changed. All the list-copying methods we used were *shallow* (not a value judgment, just a depth one).

To fix this, we need to use the `deepcopy()` function. It's in Python's standard library, but is not a built-in (base) feature, so we need to import it (see Chapter 12 for more than you ever wanted to know about imports):

```
>>> import copy
>>> a = [1, 2, [8, 9]]
>>> b = copy.deepcopy(a)
>>> a
[1, 2, [8, 9]]
>>> b
[1, 2, [8, 9]]
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 9]]
```

The `deepcopy()` function can handle deeply nested lists, dictionaries, and other objects.

Compare Lists

You can directly compare lists with the comparison operators like `==`, `<`, and so on. The operators walk through both lists, comparing elements at the same offsets. If list `a` is shorter than list `b`, and all its elements are equal, `a` is less than `b`:

```
>>> a = [7, 2]
>>> b = [7, 2, 9]
>>> a == b
False
>>> a <= b
True
>>> a < b
True
```

Iterate with for and in

In Chapter 7, you saw how to iterate over a string with `for`, but iterating over lists is much more common:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     print(cheese)
...
brie
gjetost
havarti
```

As before, `break` ends the `for` loop, and `continue` steps to the next iteration:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     if cheese.startswith('g'):
...         print("I won't eat anything that starts with 'g'")
...         break
...     else:
...         print(cheese)
...
brie
I won't eat anything that starts with 'g'
```

You can still use the optional `else` if the `for` completes without a `break`:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     if cheese.startswith('x'):
...         print("I won't eat anything that starts with 'x'")
...         break
...     else:
...         print(cheese)
... else:
...     print("Didn't find anything that started with 'x'")
...
brie
gjetost
havarti
Didn't find anything that started with 'x'
```

If the initial `for` never ran, control goes to the `else` also:

```
>>> cheeses = []
>>> for cheese in cheeses:
...     print('This shop has some lovely', cheese)
...     break
... else:  # no break means no cheese
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

Because the `cheeses` list is empty in this example, `for cheese in cheeses` never completes a single loop and its `break` statement is never executed.

Iterate Multiple Sequences with zip()

Python has more nice iteration tricks: iterating over multiple sequences in parallel by using the `zip()` function:

```
>>> days = ['Monday', 'Tuesday', 'Wednesday']
>>> fruits = ['banana', 'orange', 'peach']
>>> drinks = ['coffee', 'tea', 'beer']
>>> desserts = ['tiramisu', 'ice cream', 'pie', 'pudding']
```

```
>>> for day, fruit, drink, dessert in zip(days, fruits, drinks, desserts):
...     print(day, ": drink", drink, "- eat", fruit, "- enjoy", dessert)
...
Monday : drink coffee - eat banana - enjoy tiramisu
Tuesday : drink tea - eat orange - enjoy ice cream
Wednesday : drink beer - eat peach - enjoy pie
```

The `zip()` function stops when the shortest sequence is done. One of the lists (`desserts`) is longer than the others, so no one gets any pudding unless we extend the other lists. Read "Iterate Multiple Sequences with zip_longest()" on page 118 for a new function that gives you more control if the sequences have different lengths.

Chapter 9 shows you how the `dict()` function can create dictionaries from two-item sequences like tuples, lists, or strings. You can use `zip()` to walk through multiple sequences and make tuples from items at the same offsets. Let's make two tuples of corresponding English and French words:

```
>>> english = 'Monday', 'Tuesday', 'Wednesday'
>>> french = 'Lundi', 'Mardi', 'Mercredi'
```

Now, use `zip()` to pair these tuples. The value returned by `zip()` is itself not a tuple or list, but an iterable value that can be turned into one:

```
>>> list( zip(english, french) )
[('Monday', 'Lundi'), ('Tuesday', 'Mardi'), ('Wednesday', 'Mercredi')]
```

Feed the result of `zip()` directly to `dict()`, and voilà: a tiny English-French dictionary:

```
>>> dict( zip(english, french) )
{'Monday': 'Lundi', 'Tuesday': 'Mardi', 'Wednesday': 'Mercredi'}
```

Iterate Multiple Sequences with zip_longest()

As you just saw, plain `zip()` runs out of gas when the shortest sequence ends. But a variant keeps going until the longest sequence ends. The `zip_longest()` function will supply `None` unless you provide a `fillvalue` for the unmatched items at the end of the shorter sequence:

```
>>> from itertools import zip_longest
>>> a = [1, 2, 3]
>>> b = [1, 2, 3, 4, 5]
>>> for x in zip(a, b):
...     print(x)
...
(1, 1)
(2, 2)
(3, 3)
>>> for x in zip_longest(a, b):
...     print(x)
...
```

```
(1, 1)
(2, 2)
(3, 3)
(None, 4)
(None, 5)
>>> for x in zip_longest(a, b, fillvalue='!'):
...     print(x)
...
(1, 1)
(2, 2)
(3, 3)
('!', 4)
('!', 5)
```

Create a List with a Comprehension

You saw how to create a list with square brackets or the `list()` function. Here, we look at how to create a list with a *list comprehension*, which incorporates the `for`/`in` iteration that you just saw.

You could build a list of integers from `1` to `5`, one item at a time, like this:

```
>>> number_list = []
>>> number_list.append(1)
>>> number_list.append(2)
>>> number_list.append(3)
>>> number_list.append(4)
>>> number_list.append(5)
>>> number_list
[1, 2, 3, 4, 5]
```

Or you could also use an iterator and the `range()` function:

```
>>> number_list = []
>>> for number in range(1, 6):
...     number_list.append(number)
...
>>> number_list
[1, 2, 3, 4, 5]
```

Or you could just turn the output of `range()` into a list directly:

```
>>> number_list = list(range(1, 6))
>>> number_list
[1, 2, 3, 4, 5]
```

All these approaches are valid Python code and will produce the same result. However, a more Pythonic (and often faster) way to build a list is by using a *list comprehension*. The simplest form of list comprehension looks like this:

```
[expression for item in iterable]
```

Here's how a list comprehension would build the integer list:

```
>>> number_list = [number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

In the first line, you need the first `number` variable to produce values for the list: that is, to put a result of the loop into `number_list`. The second `number` is part of the `for` loop. To show that the first `number` is an expression, try this variant:

```
>>> number_list = [number-1 for number in range(1,6)]
>>> number_list
[0, 1, 2, 3, 4]
```

The list comprehension moves the loop inside the square brackets. This comprehension example really wasn't simpler than the previous example, but there's more that you can do. A list comprehension can include a conditional expression, looking something like this:

```
[expression for item in iterable if condition]
```

Let's make a new comprehension that builds a list of only the odd numbers from `1` to `5` (remember that `number % 2` is `True` for odd numbers and `False` for even numbers):

```
>>> a_list = [number for number in range(1,6) if number % 2 == 1]
>>> a_list
[1, 3, 5]
```

Now, the comprehension is a little more compact than its traditional counterpart:

```
>>> a_list = []
>>> for number in range(1,6):
...     if number % 2 == 1:
...         a_list.append(number)
...
>>>  a_list
[1, 3, 5]
```

Finally, just as there can be nested loops, there can be more than one set of `for` clauses in the corresponding comprehension. To show this, let's first try a plain old nested loop and print the results:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> for row in rows:
...     for col in cols:
...         print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

Now, let's use a comprehension and assign it to the variable `cells`, making it a list of `(row, col)` tuples:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> cells = [(row, col) for row in rows for col in cols]
>>> for cell in cells:
...     print(cell)
...
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

By the way, you can also use *tuple unpacking* to get the `row` and `col` values from each tuple as you iterate over the `cells` list:

```
>>> for row, col in cells:
...     print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

The `for row` and `for col` fragments in the list comprehension could also have had their own `if` tests.

Create Lists of Lists

Lists can contain elements of different types, including other lists, as illustrated here:

```
>>> small_birds = ['hummingbird', 'finch']
>>> extinct_birds = ['dodo', 'passenger pigeon', 'Norwegian Blue']
>>> carol_birds = [3, 'French hens', 2, 'turtledoves']
>>> all_birds = [small_birds, extinct_birds, 'macaw', carol_birds]
```

So what does `all_birds`, a list of lists, look like?

```
>>> all_birds
[['hummingbird', 'finch'], ['dodo', 'passenger pigeon', 'Norwegian Blue'],
'macaw', [3, 'French hens', 2, 'turtledoves']]
```

Let's look at the first item in it:

```
>>> all_birds[0]
['hummingbird', 'finch']
```

The first item is a list: in fact, it's `small_birds`, the first item we specified when creating `all_birds`. You should be able to guess the second item:

```
>>> all_birds[1]
['dodo', 'passenger pigeon', 'Norwegian Blue']
```

It's the second item we specified, `extinct_birds`. If we want the first item of `extinct_birds`, we can extract it from `all_birds` by specifying two indices:

```
>>> all_birds[1][0]
'dodo'
```

The `[1]` refers to the list that's the second item in `all_birds`, and the `[0]` refers to the first item in that inner list.

Tuples Versus Lists

You can often use tuples in place of lists, but they have fewer functions (there is no `append()`, `insert()`, and so on) because tuples can't be modified after creation. Why not just use lists instead of tuples everywhere?

- Tuples use less space.
- You can't clobber tuple items by mistake.
- You can use tuples as dictionary keys (see Chapter 9).
- *Named tuples* (see "Named Tuples" on page 209) can be a simple alternative to objects.

I won't go into much more detail about tuples here. In everyday programming, you'll use lists and dictionaries more.

There Are No Tuple Comprehensions

Mutable types (lists, dictionaries, and sets) have comprehensions. Immutable types like strings and tuples need to be created with the other methods listed in their sections.

You might have thought that changing the square brackets of a list comprehension to parentheses would create a tuple comprehension. And that change would appear to work because no exception occurs if you type this:

```
>>> number_thing = (number for number in range(1, 6))
```

The thing within the parentheses is something else entirely: a *generator comprehension*, and it returns a *generator object*:

```
>>> type(number_thing)
<class 'generator'>
```

I'll get into generators in more detail in "Generators" on page 168. A generator is one way to provide data to an iterator.

Review/Preview

Tuples and lists are sequences, and you access their elements by their relative positions. Lists can also be changed in place (mutable), but tuples can't (immutable). Sequences are a type of iterable that you can step through with `for` and `in`.

The next chapter is about sets and dictionaries, which emphasize the values of their elements rather than their positions in a sequence.

Practice

Use lists and tuples with numbers (Chapter 3) and strings (Chapter 4) to represent elements in the real world with great variety.

8.1 Create a list called `years_list`, starting with the year of your birth, and each year thereafter until the year of your fifth birthday. For example, if you were born in 1980, the list would be `years_list = [1980, 1981, 1982, 1983, 1984, 1985]`. If you're fewer than five years old and reading this book, I don't know what to tell you.

8.2 In which year in `years_list` was your third birthday? Remember, you were 0 years of age for your first year.

8.3 In which year in `years_list` were you the oldest?

8.4 Make and print a list called `things` with these three strings as elements: `mozzarella`, `cinderella`, `salmonella`.

8.5 Capitalize the element in `things` that refers to a person and then print the list. Did it change the element in the list?

8.6 Make the cheesy element of `things` all uppercase and then print the list.

8.7 Delete the disease element from `things`, collect your Nobel Prize, and then print the list.

8.8 Create a list called `surprise` with the elements `Groucho`, `Chico`, and `Harpo`.

8.9 Lowercase the last element of the `surprise` list, reverse it, and then capitalize it.

8.10 Use a list comprehension to make a list called `even` of the even numbers in `range(10)`.

8.11 Let's create a jump-rope rhyme maker. You'll print a series of two-line rhymes. Start with this program fragment:

```
start1 = ["fee", "fie", "foe"]
rhymes = [
    ("flop", "get a mop"),
    ("fope", "turn the rope"),
    ("fa", "get your ma"),
    ("fudge", "call the judge"),
    ("fat", "pet the cat"),
    ("fog", "walk the dog"),
    ("fun", "say we're done"),
    ]
start2 = "Someone better"
```

Do the following for each tuple (`first`, `second`) in `rhymes`.

For the first line:

- Print each string in `start1`, capitalized and followed by an exclamation point and a space.
- Print `first`, also capitalized and followed by an exclamation point.

For the second line:

- Print `start2` and a space.
- Print `second` and a period.

8.12 Print each list question with its correctly matching answer, in this form:

Q: *question*
A: *answer*

```
>>> questions = [
...     "We don't serve strings around here. Are you a string?",
...     "What is said on Father's Day in the forest?",
...     "What makes the sound 'Sis! Boom! Bah!'?"
...     ]
>>> answers = [
...     "An exploding sheep.",
...     "No, I'm a frayed knot.",
...     "'Pop!' goes the weasel."
...     ]
```

CHAPTER 9

Dictionaries and Sets

If a word in the dictionary were misspelled, how would we know?

—Steven Wright

Dictionaries are particularly useful key-value data structures, and you'll see them used everywhere in Python. Sets are just bags of unique keys and are often very handy.

Dictionaries

A *dictionary* is similar to a list, but the order of items doesn't matter, and they aren't selected by an offset such as 0 or 1. Instead, you specify a unique *key* to associate with each *value*. This key is often a string, but it can be any of Python's immutable types: Boolean, integer, float, tuple, string, and others—even ones that you've defined. Dictionaries are mutable, so you can add, delete, and change their key-value elements. If you've worked with languages that support only arrays or lists, you'll love dictionaries.

In other languages, dictionaries might be called *associative arrays*, *hashes*, or *hashmaps*. In Python, a dictionary is also called a *dict* to save syllables and make teenage boys snicker.

Create with {}

To create a dictionary, you place curly braces ({}) around comma-separated *`key : value`* pairs. The simplest dictionary is empty, containing no keys or values at all:

```
>>> empty_dict = {}
>>> empty_dict
{}
```

Let's make a small dictionary with quotes from Ambrose Bierce's *The Devil's Dictionary*:

```
>>> bierce = {
...     "day": "A period of twenty-four hours, mostly misspent",
...     "positive": "Mistaken at the top of one's voice",
...     "misfortune": "The kind of fortune that never misses",
...     }
>>>
```

Typing the dictionary's name in the interactive interpreter will print its keys and values:

```
>>> bierce
{'day': 'A period of twenty-four hours, mostly misspent',
'positive': "Mistaken at the top of one's voice",
'misfortune': 'The kind of fortune that never misses'}
```

In Python, it's OK to leave a comma after the last item of a list, tuple, or dictionary. Also, you don't need to indent, as I did in the previous example, when you're typing keys and values within the curly braces. The indentation just helps readability.

Create with dict()

Some people don't like typing so many curly braces and quotes. You can also create a dictionary by passing named arguments and values to the `dict()` function.

Here's the traditional way:

```
>>> acme_customer = {'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
>>> acme_customer
{'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
```

And this way uses `dict()`:

```
>>> acme_customer = dict(first="Wile", middle="E", last="Coyote")
>>> acme_customer
{'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
```

One limitation of the second way is that the argument names need to be legal variable names (no spaces, no reserved words):

```
>>> x = dict(name="Elmer", def="hunter")
  File "<stdin>", line 1
    x = dict(name="Elmer", def="hunter")
                           ^
SyntaxError: invalid syntax
```

Convert with dict()

You can also use the `dict()` function to convert two-value sequences into a dictionary. You might run into such key-value sequences at times, such as "Strontium, 90, Carbon, 14."[1] The first item in each sequence is used as the key and the second as the value.

First, here's a small example using `lol` (a list of two-item lists):

```
>>> lol = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(lol)
{'a': 'b', 'c': 'd', 'e': 'f'}
```

We could have used any sequence containing two-item sequences. Here are other examples.

A list of two-item tuples:

```
>>> lot = [ ('a', 'b'), ('c', 'd'), ('e', 'f') ]
>>> dict(lot)
{'a': 'b', 'c': 'd', 'e': 'f'}
```

A tuple of two-item lists:

```
>>> tol = ( ['a', 'b'], ['c', 'd'], ['e', 'f'] )
>>> dict(tol)
{'a': 'b', 'c': 'd', 'e': 'f'}
```

A list of two-character strings:

```
>>> los = [ 'ab', 'cd', 'ef' ]
>>> dict(los)
{'a': 'b', 'c': 'd', 'e': 'f'}
```

A tuple of two-character strings:

```
>>> tos = ( 'ab', 'cd', 'ef' )
>>> dict(tos)
{'a': 'b', 'c': 'd', 'e': 'f'}
```

"Iterate Multiple Sequences with zip()" on page 117 introduced you to the `zip()` function, which makes it easy to create these two-item sequences.

1 Also, the final score in the Strontium-Carbon game. Strontium is hot this year.

Add or Change an Item by [key]

Adding an item to a dictionary is easy. Just refer to the item by its key and assign a value. If the key is already present in the dictionary, the existing value is replaced by the new one. If the key is new, it's added to the dictionary with its value. Unlike with lists, you don't need to worry about Python throwing an exception during assignment by specifying an index that's out of range.

Let's make a dictionary of most of the members of Monty Python, using their last names as keys, and first names as values:

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Idle': 'Eric',
'Jones': 'Terry', 'Palin': 'Michael'}
```

We're missing one member: the one born in America, Terry Gilliam. An anonymous programmer attempts to add him but botches the first name:

```
>>> pythons['Gilliam'] = 'Gerry'
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Idle': 'Eric',
'Jones': 'Terry', 'Palin': 'Michael', 'Gilliam': 'Gerry'}
```

And here's some repair code by another programmer who is Pythonic in more than one way:

```
>>> pythons['Gilliam'] = 'Terry'
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Idle': 'Eric',
'Jones': 'Terry', 'Palin': 'Michael', 'Gilliam': 'Terry'}
```

By using the same key (`Gilliam`), we replace the original value `Gerry` with `Terry`.

Remember that dictionary keys must be *unique*. That's why we used last names for keys instead of first names here; two members of Monty Python have the first name `Terry`! If you use a key more than once, the last value wins:

```
>>> some_pythons = {
...     'Graham': 'Chapman',
...     'John': 'Cleese',
...     'Eric': 'Idle',
...     'Terry': 'Gilliam',
...     'Michael': 'Palin',
...     'Terry': 'Jones',
...     }
```

```
>>> some_pythons
{'Graham': 'Chapman', 'John': 'Cleese', 'Eric': 'Idle',
'Terry': 'Jones', 'Michael': 'Palin'}
```

We first assign the value `Gilliam` to the key `Terry` and then replace it with the value `Jones`.

One last restriction on dictionary keys: they need to be *hashable*. What? Essentially, this means that the key's contents cannot change. So you can use a tuple as a key, but not a list:

```
>>> key  = (1, 2)
>>> val = "a"
>>> d = {key: val}
>>> key = [1, 2]
>>> d = {key: val}
Traceback (most recent call last):
  File "<python-input-15>", line 1, in <module>
    d = {key: val}
        ^^^^^^^^^^
TypeError: unhashable type: 'list'
```

Get an Item by [key] or with get()

Getting a single element is the most common use of a dictionary. You specify the dictionary and key to get the corresponding value. Use the `some_pythons` dictionary from the previous section and get the value associated with the key `John`:

```
>>> some_pythons['John']
'Cleese'
```

If the key is not present in the dictionary, you'll get an exception:

```
>>> some_pythons['Groucho']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Groucho'
```

There are two good ways to avoid this. The first is to test for the key at the outset by using `in`, as you saw in the previous section:

```
>>> 'Groucho' in some_pythons
False
```

The second is to use the special dictionary `get()` function. You provide the dictionary, key, and an optional value. If the key exists, you get its value:

```
>>> some_pythons.get('John')
'Cleese'
```

If not, you get the optional value, if you specified one:

```
>>> some_pythons.get('Groucho', 'Not a Python')
'Not a Python'
```

Otherwise, you get `None` (which displays nothing in the interactive interpreter):

```
>>> some_pythons.get('Groucho')
>>>
```

Iterate with for and in

Iterating over a dictionary returns the keys, in the order they were originally defined or added. In this example, the keys are the types of cards in the board game Clue (Cluedo outside of North America):

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
...               'person': 'Col. Mustard'}
>>> for card in accusation:
...     print(card)
...
room
weapon
person
```

Using the dictionary's `keys()` method does the same thing:

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
...     'person': 'Col. Mustard'}
>>> for card in accusation.keys():
...     print(card)
...
room
weapon
person
```

In Python 3, `keys()` returns `dict_keys()`, which is an iterable view of the keys that works with `for ... in`. This approach is handy with large dictionaries because it doesn't use the time and memory to create and store a list that you might not use. But often you *do* want to save all the keys in a list. In Python 3, you need to call `list()` to convert a `dict_keys` object to a list.

```
>>> card_list = list(accusation.keys())
>>> card_list
['room', 'weapon', 'person']
```

To iterate over the values rather than the keys, use the dictionary's `values()` function:

```
>>> for value in accusation.values():
...     print(value)
...
ballroom
lead pipe
Col. Mustard
```

To return both the key and value as a tuple, you can use the `items()` function:

```
>>> for item in accusation.items():
...     print(item)
...
('room', 'ballroom')
('weapon', 'lead pipe')
('person', 'Col. Mustard')
```

You can assign the pair of values returned in each step to a tuple. For each tuple returned by `items()`, assign the first value (the key) to `card`, and the second (the value) to `contents`:

```
>>> for card, contents in accusation.items():
...     print('Card', card, 'has the contents', contents)
...
Card weapon has the contents lead pipe
Card person has the contents Col. Mustard
Card room has the contents ballroom
```

Get Length with len()

Count your key-value pairs:

```
>>> len(signals)
3
```

Combine/update dicts

You can merge dictionaries in more than one way, although this is arguably contrary to a dictum from the Zen of Python: *There should be one—and preferably only one—obvious way to do it*.

Use update()

You can use the `update()` method to copy the keys and values of one dictionary into another.

Let's define the `pythons` dictionary, with all members:

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
```

```
...     'Gilliam': 'Terry',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Gilliam': 'Terry',
'Idle': 'Eric', 'Jones': 'Terry', 'Palin': 'Michael'}
```

We also have a dictionary of other humorous persons called `others`:

```
>>> others = { 'Marx': 'Groucho', 'Howard': 'Moe' }
```

Now, along comes another anonymous programmer who decides that the members of `others` should be members of Monty Python:

```
>>> pythons.update(others)
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Gilliam': 'Terry',
'Idle': 'Eric', 'Jones': 'Terry', 'Palin': 'Michael',
'Marx': 'Groucho', 'Howard': 'Moe'}
```

What happens if the second dictionary has the same key as the dictionary into which it's being merged? The value from the second dictionary wins:

```
>>> first = {'a': 1, 'b': 2}
>>> second = {'b': 'platypus'}
>>> first.update(second)
>>> first
{'a': 1, 'b': 'platypus'}
```

Use {**a, **b}

Starting with version 3.5, Python has a new way to merge dictionaries, using the ** unicorn glitter (or *dictionary unpacking*), which has a very different use in Chapter 10:

```
>>> first = {'a': 'agony', 'b': 'bliss'}
>>> second = {'b': 'bagels', 'c': 'candy'}
>>> {**first, **second}
{'a': 'agony', 'b': 'bagels', 'c': 'candy'}
```

You can pass more than two dictionaries:

```
>>> third = {'d': 'donuts'}
>>> {**first, **third, **second}
{'a': 'agony', 'b': 'bagels', 'd': 'donuts', 'c': 'candy'}
```

These are *shallow* copies. See "Copy Everything with deepcopy()" on page 135 if you want full copies of the keys and values, with no connection to their origin dictionaries.

Use |

Python 3.9 added the ability to use the operator |. Adapting one of the earlier examples, we can use | as follows::

```
>>> first = {'a': 1, 'b': 2}
>>> second = {'b': 'platypus'}
>>> first | second
{'a': 1, 'b': 'platypus'}
>>> first
{'a': 1, 'b': 2}
>>> first |= second
>>> first
{'a': 1, 'b': 'platypus'}
```

In many ways, this is the best approach. Using `update()` always modifies the target string in place, and the bang-bang (`**`) method isn't very intuitive.

Delete an Item by Key with del

The earlier `pythons.update(others)` code from our anonymous programmer is technically correct but factually wrong. The members of `others`, although funny and famous, were not in Monty Python. Let's undo those last two additions:

```
>>> del pythons['Marx']
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Gilliam': 'Terry',
'Idle': 'Eric', 'Jones': 'Terry', 'Palin': 'Michael',
'Howard': 'Moe'}
>>> del pythons['Howard']
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Gilliam': 'Terry',
'Idle': 'Eric', 'Jones': 'Terry', 'Palin': 'Michael'}
```

Get an Item by Key and Delete It with pop(key)

This approach combines `get()` and `del`. You give `pop()` a key; if it exists in the dictionary, `pop()` returns the matching value and deletes the key-value pair. If the key doesn't exist, Python raises an exception:

```
>>> len(pythons)
6
>>> pythons.pop('Palin')
'Michael'
>>> len(pythons)
5
>>> pythons.pop('Palin')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Palin'
```

But if you give `pop()` a second default argument (as with `get()`), all is well and the dictionary is not changed:

```
>>> pythons.pop('First', 'Hugo')
'Hugo'
>>> len(pythons)
5
```

Delete All Items with clear()

To delete all keys and values from a dictionary, use `clear()` or just reassign an empty dictionary (`{}`) to the name:

```
>>> pythons.clear()
>>> pythons
{}
>>> pythons = {}
>>> pythons
{}
```

Test for a Key with in

If you want to know whether a key exists in a dictionary, use `in`. Let's redefine the `pythons` dictionary again, this time omitting a name or two:

```
>>> pythons = {'Chapman': 'Graham', 'Cleese': 'John',
... 'Jones': 'Terry', 'Palin': 'Michael', 'Idle': 'Eric'}
```

Now let's see who's in there:

```
>>> 'Chapman' in pythons
True
>>> 'Palin' in pythons
True
```

Did we remember to add Terry Gilliam this time?

```
>>> 'Gilliam' in pythons
False
```

Drat.

Assign with =

As with lists, any change you make to a dictionary will be reflected in all the names that refer to that dictionary:

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': 'smile for the camera'}
>>> save_signals = signals
>>> signals['blue'] = 'confuse everyone'
>>> save_signals
```

```
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera',
'blue': 'confuse everyone'}
```

This is another example of the point I made in Chapter 2. You can have more than one name (variable) pointing to the same value, so changing a value by one name changes it for the others.

Copy with copy()

To copy keys and values from a dictionary to another dictionary and avoid modifying the first dictionary, you can use `copy()`:

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': 'smile for the camera'}
>>> original_signals = signals.copy()
>>> signals['blue'] = 'confuse everyone'
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera',
'blue': 'confuse everyone'}
>>> original_signals
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera'}
>>>
```

This is a *shallow* copy, which works if the dictionary values are immutable (as they are in this case). If they aren't, you need `deepcopy()`.

Copy Everything with deepcopy()

Suppose that the value for `red` in the previous example is a list instead of a single string:

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': ['stop', 'smile']}
>>> signals_copy = signals.copy()
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'smile']}
>>> signals_copy
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'smile']}
>>>
```

Let's change one of the values in the `red` list:

```
>>> signals['red'][1] = 'sweat'
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'sweat']}
>>> signals_copy
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'sweat']}
```

You get the usual Python change-by-either-name behavior. The `copy()` method copies the values as is, meaning `signal_copy` gets the same list value for `red` that `signals` has.

The solution is `deepcopy()`, which makes true copies all the way down:

```
>>> import copy
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': ['stop', 'smile']}
>>> signals_copy = copy.deepcopy(signals)
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'smile']}
>>> signals_copy
{'green': 'go',
'yellow':'go faster',
'red': ['stop', 'smile']}
>>> signals['red'][1] = 'sweat'
>>> signals
{'green': 'go',
'yellow': 'go faster',
red': ['stop', 'sweat']}
>>> signals_copy
{'green': 'go',
'yellow': 'go faster',
red': ['stop', 'smile']}
```

Compare Dictionaries

Much like lists and tuples in Chapter 8, dictionaries can be compared with the simple comparison operators == and !=:

```
>>> a = {1:1, 2:2, 3:3}
>>> b = {3:3, 1:1, 2:2}
>>> a == b
True
```

Other operators won't work:

```
>>> a = {1:1, 2:2, 3:3}
>>> b = {3:3, 1:1, 2:2}
>>> a <= b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<=' not supported between instances of 'dict' and 'dict'
```

Python compares the keys and values one by one. The order in which they were originally created doesn't matter. In this example, `a` and `b` are almost equal, except key `1` has the list value `[1, 2]` in `a` and the list value `[1, 1]` in `b`:

```
>>> a = {1: [1, 2], 2: [1], 3:[1]}
>>> b = {1: [1, 1], 2: [1], 3:[1]}
>>> a == b
False
```

Use Dictionary Comprehensions

Not to be outdone by those bourgeois lists, dictionaries also have comprehensions. The simplest form looks familiar:

```
{key_expression : value_expression for expression in iterable}
```

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in word}
>>> letter_counts
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

We're running a loop over each of the seven letters in the string `letters` and counting the number of times that letter appears. Two uses of `word.count(letter)` are a waste of time because we have to count all the `e`'s twice and all the `t`'s twice. But when we count the `e`'s the second time, we do no harm because we just replace the entry in the dictionary that was already there; the same goes for counting the `t`'s. So, the following would have been a teeny bit more Pythonic:[2]

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in set(word)}
>>> letter_counts
{'t': 2, 'l': 1, 'e': 2, 'r': 1, 's': 1}
```

The dictionary's keys are in a different order than the previous example because iterating `set(word)` returns letters in a different order than iterating the string `word`.

2 See "Count Items with Counter()" on page 225 for an even more Pythonicalistic solution.

Similar to list comprehensions, dictionary comprehensions can also have `if` tests and multiple `for` clauses:

```
{key_expression : value_expression for expression in iterable if condition}
```

```
>>> vowels = 'aeiou'
>>> word = 'onomatopoeia'
>>> vowel_counts = {letter: word.count(letter) for letter in set(word)
        if letter in vowels}
>>> vowel_counts
{'e': 1, 'i': 1, 'o': 4, 'a': 2}
```

See PEP-274 for more examples of dictionary comprehensions.

Sets

A *set* is like a dictionary with its values thrown away, leaving only the keys. As with a dictionary, each key must be unique. You use a set when you want to know only that something exists and nothing else about it. A set is a bag of keys. Use a dictionary if you want to attach some information to the key as a value.

Create with set() or {}

Create an empty set with `set()`:

```
>>> empty = set()
>>> empty
set()
```

You can also create sets by surrounding an iterable data type with curly braces (`{}`). But the set can't be empty:

```
>>> empty = {}
>>> empty
{}
>>> type(empty)
<class 'dict'>
```

Why do dicts win the empty definition war? Dicts got there first, and possession is 90% of the law.[3]

To create a non-empty set, use `{}` with values, or use the `set()` function with an iterable argument (a tuple, list, string, or dict):

```
>>> evens = {0, 2, 4, 6, 8}
>>> evens
{0, 2, 4, 6, 8}
>>> evens = set( [0, 2, 4, 6, 8] )
```

3 According to lawyers and exorcists.

```
>>> evens
{0, 2, 4, 6, 8}
>>> evens = set( (0, 2, 4, 6, 8) )
>>> evens
{0, 2, 4, 6, 8}
>>> evens = set( {0:1, 2:4, 4:8, 6:12, 8:16} )
>>> evens
{0, 2, 4, 6, 8}
```

Sets contain unique values, so any duplicates are dropped:

```
>>> set( 'letters' )
{'l', 'r', 's', 't', 'e'}
```

Notice that `set()` doesn't accept multiple values, as {} does:

```
>>> evens = set(0, 2, 4, 6, 8)
Traceback (most recent call last):
  File "<python-input-37>", line 1, in <module>
    evens = set(0, 2, 4, 6, 8)
TypeError: set expected at most 1 argument, got 5
```

If you do give `set()` a single-item tuple, you need parentheses and the following comma:

```
>>> x = set( (1) )
Traceback (most recent call last):
  File "<python-input-39>", line 1, in <module>
    x = set( (1) )
TypeError: 'int' object is not iterable
>>> x = set( 1, )
Traceback (most recent call last):
  File "<python-input-42>", line 1, in <module>
    x = set( 1, )
TypeError: 'int' object is not iterable
>>> x = set( (1,) )
>>> x
{1}
```

Sets are unordered, so iterating over one is like dumping a bag of values.

Get Length with len()

Let's count our reindeer:

```
>>> reindeer = set( ['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'] )
>>> len(reindeer)
4
```

Add an Item with add()

Throw another item into a set with the set `add()` method:

```
>>> s = set((1,2,3))
>>> s
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

If you try to add something that was already in there, no problem:

```
>>> s = set((1,2,3))
>>> s.add(2)
>>> s
{1, 2, 3}
```

Delete an Item with remove()

You can delete a value from a set by value:

```
>>> s = set((1,2,3))
>>> s.remove(3)
>>> s
{1, 2}
```

Combine with |

Combine sets with `|`:

```
>>> a = set((1, 3, 5))
>>> b = set((2, 4, 6))
>>> a | b
{1, 2, 3, 4, 5, 6}
```

Iterate with for and in

As with dictionaries, you can iterate over all items in a set:

```
>>> furniture = set(('sofa', 'ottoman', 'table'))
>>> for piece in furniture:
...     print(piece)
...
ottoman
table
sofa
```

Test for a Value with in

The most common use of a set is to test for a value via `in`. We'll make a dictionary called `drinks`. Each key is the name of a mixed drink, and the corresponding value is a set of that drink's ingredients:

```
>>> drinks = {
...     'martini': {'vodka', 'vermouth'},
...     'black russian': {'vodka', 'kahlua'},
...     'white russian': {'cream', 'kahlua', 'vodka'},
...     'manhattan': {'rye', 'vermouth', 'bitters'},
...     'screwdriver': {'orange juice', 'vodka'}
...     }
```

Even though both are enclosed by curly braces ({ and }), a set is just a bunch of values, and a dictionary contains key-value pairs.

Which drinks contain vodka?

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents:
...         print(name)
...
screwdriver
martini
black russian
white russian
```

We want something with vodka but are lactose intolerant and think vermouth tastes like kerosene:

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not ('vermouth' in contents or
...         'cream' in contents):
...         print(name)
...
screwdriver
black russian
```

We'll rewrite this a bit more succinctly in the next section.

Use Combinations and Operators

At some bygone time, in some places, set theory was taught in elementary school along with basic mathematics. If your school skipped it (or you were staring out the window), Figure 9-1 shows the ideas of set union and intersection.

Suppose that you want to merge two sets. Because a set must contain unique keys, the *union* of two sets will contain only one of each key. The *null* or *empty* set is a set with zero elements. In Figure 9-1, an example of a null set would be female names beginning with `X`.

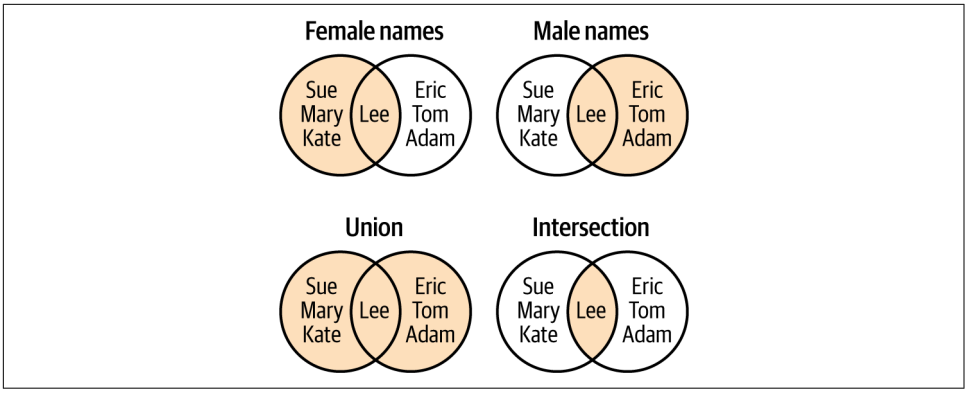

Figure 9-1. Common things to do with sets

What if you want to check for combinations of set values? Suppose that you want to find any drink that has orange juice or vermouth? Let's use the *set intersection operator*, which is an ampersand (`&`):

```
>>> for name, contents in drinks.items():
...     if contents & {'vermouth', 'orange juice'}:
...         print(name)
...
screwdriver
martini
manhattan
```

The result of the `&` operator is a set that contains all the items appearing in both sets that you compare. If neither of those ingredients are in `contents`, the `&` returns an empty set, which is considered `False`.

Now, let's rewrite the example from the previous section, in which we want vodka but neither cream nor vermouth:

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not contents & {'vermouth', 'cream'}:
...         print(name)
...
screwdriver
black russian
```

Let's save the ingredient sets for these two drinks in variables, just to save our delicate fingers some typing in the coming examples:

```
>>> bruss = drinks['black russian']
>>> wruss = drinks['white russian']
```

The following are examples of all the set operators. Some have special punctuation, some have special functions, and some have both. Let's use test sets `a` (contains `1` and `2`) and `b` (contains `2` and `3`):

```
>>> a = {1, 2}
>>> b = {2, 3}
```

As you saw earlier, you get the *intersection* (members common to both sets) with the special punctuation symbol `&`. The set `intersection()` function does the same:

```
>>> a & b
{2}
>>> a.intersection(b)
{2}
```

This snippet uses our saved drink variables:

```
>>> bruss & wruss
{'kahlua', 'vodka'}
```

In this example, get the *union* (members of either set) by using `|` or the set `union()` function:

```
>>> a | b
{1, 2, 3}
>>> a.union(b)
{1, 2, 3}
```

And here's the alcoholic version:

```
>>> bruss | wruss
{'cream', 'kahlua', 'vodka'}
```

The *difference* (members of the first set but not the second) is obtained by using the hyphen character (`-`) or the `difference()` function:

```
>>> a - b
{1}
>>> a.difference(b)
{1}
>>> bruss - wruss
set()
>>> wruss - bruss
{'cream'}
```

By far, the most common set operations are union, intersection, and difference. I've included the others for completeness in the examples that follow, but you might never use them.

The *exclusive or* (items in one set or the other, but not both) uses `^` or `symmetric_difference()`:

```
>>> a ^ b
{1, 3}
>>> a.symmetric_difference(b)
{1, 3}
```

This finds the exclusive ingredient in our two Russian drinks:

```
>>> bruss ^ wruss
{'cream'}
```

You can check whether one set is a *subset* of another (all members of the first set are also in the second set) by using `<=` or `issubset()`:

```
>>> a <= b
False
>>> a.issubset(b)
False
```

Adding cream to a Black Russian makes a White Russian, so `wruss` is a superset of `bruss`:

```
>>> bruss <= wruss
True
```

Is any set a subset of itself? Yup:[4]

```
>>> a <= a
True
>>> a.issubset(a)
True
```

To be a *proper subset*, the second set needs to have all the members of the first and more. Calculate it by using `<`, as in this example:

```
>>> a < b
False
>>> a < a
False
>>> bruss < wruss
True
```

A *superset* is the opposite of a subset (all members of the second set are also members of the first). This uses `>=` or `issuperset()`:

```
>>> a >= b
False
>>> a.issuperset(b)
False
>>> wruss >= bruss
True
```

Any set is a superset of itself:

```
>>> a >= a
True
>>> a.issuperset(a)
True
```

4 Although, paraphrasing Groucho Marx, "I wouldn't want to belong to a club that would have someone like me as a member."

And finally, you can find a *proper superset* (the first set has all members of the second, and more) by using >, as shown here:

```
>>> a > b
False
>>> wruss > bruss
True
```

You can't be a proper superset of yourself:

```
>>> a > a
False
```

Create Set Comprehensions

No one wants to be left out, so even sets have comprehensions. The simplest version looks like the list and dictionary comprehensions that you've just seen:

```
{ expression for expression in iterable }
```

And it can have the optional condition tests:

```
{ expression for expression in iterable if condition }
```

```
>>> a_set = {number for number in range(1,6) if number % 3 == 1}
>>> a_set
{1, 4}
```

Create an Immutable Set with frozenset()

If you want to create a set that can't be changed, call the `frozenset()` function with any iterable argument:

```
>>> frozenset([3, 2, 1])
frozenset({1, 2, 3})
>>> frozenset(set([2, 1, 3]))
frozenset({1, 2, 3})
>>> frozenset({3, 1, 2})
frozenset({1, 2, 3})
>>> frozenset( (2, 3, 1) )
frozenset({1, 2, 3})
```

Is it really frozen?

```
>>> fs = frozenset([3, 2, 1])
>>> fs
frozenset({1, 2, 3})
>>> fs.add(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Yes, pretty frosty.

Review/Preview

You'll see dictionaries everywhere in Python code. They're particularly useful in data processing, when you want to group elements by a certain value.

The next chapter goes from data structure land back to code structure land: functions wrap chunks of code, give them names, take inputs, and return outputs.

Practice

9.1 Make an English-to-French dictionary called `e2f` and print it. Here are your starter words: `dog` is `chien`, `cat` is `chat`, and `walrus` is `morse`.

9.2 Using your three-word dictionary `e2f`, print the French word for `walrus`.

9.3 Make a French-to-English dictionary called `f2e` from `e2f`. Use the `items()` method.

9.4 Print the English equivalent of the French word `chien`.

9.5 Print the set of English words from `e2f`.

9.6 Make a multilevel dictionary called `life`. Use these strings for the topmost keys: `'animals'`, `'plants'`, and `'other'`. Make the `'animals'` key refer to another dictionary with the keys `'cats'`, `'octopi'`, and `'emus'`. Make the `'cats'` key refer to a list of strings with the values `'Henri'`, `'Grumpy'`, and `'Lucy'`. Make all the other keys refer to empty dictionaries.

9.7 Print the top-level keys of `life`.

9.8 Print the keys for `life['animals']`.

9.9 Print the values for `life['animals']['cats']`.

9.10 Use a dictionary comprehension to create the dictionary `squares`. Use `range(10)` to return the keys, and use the square of each key as its value.

9.11 Use a set comprehension to create the set `odd` from the odd numbers in `range(10)`.

9.12 Use a generator comprehension to return the string `'Got '` and a number for the numbers in `range(10)`. Iterate through this by using a `for` loop.

9.13 Use `zip()` to make a dictionary from the key tuple `('optimist', 'pessimist', 'troll')` and the values tuple `('The glass is half full', 'The glass is half empty', 'How did you get a glass?')`.

9.14 Use `zip()` to make a dictionary called `movies` that pairs these lists: `titles = ['Creature of Habit', 'Crewel Fate', 'Sharks On a Plane']` and `plots = ['A nun turns into a monster', 'A haunted yarn shop', 'Check your exits']`.

CHAPTER 10
Functions

The smaller the function, the greater the management.
—C. Northcote Parkinson

So far, all our Python code examples have been little fragments. These are good for small tasks, but no one wants to retype fragments all the time. We need a way of organizing larger code into manageable pieces and making it reusable.

The first step to code reuse is the *function*: a named piece of code, separate from all others. A function can take any number and type of input *parameters* and return a *result*. The result can be as follows:

Nothing
: Python returns the value `None`.

Something
: A single object, which may have a single value, or multiple elements (e.g., a tuple or list).

You can take two actions with a function:

- *Define* it, with zero or more parameters
- *Call* it with zero or more arguments, and get a result

Define a Function with def

To define a Python function, you type `def`, the function name, parentheses enclosing any input *parameters* to the function, and then finally, a colon (`:`). Function names

have the same rules as variable names (they must start with a letter or _ and contain only letters, numbers, or _).

Let's take the definition process one step at a time, and first define and call a function that has no parameters. Here's the simplest Python function:

```
>>> def do_nothing():
...     pass
```

Why that `pass`? Well, Python won't allow an empty line there, so `pass` is needed to say, "Hey, nothing to see here."

Even for a function with no parameters like this one, you still need the parentheses and the colon in its definition. The next line needs to be indented, just as you would indent code under an `if` statement. Python requires the `pass` statement to show that this function does nothing. It's the equivalent of *This page intentionally left blank* (even though it isn't anymore).

Call a Function with Parentheses

You call this function just by typing its name and parentheses. It works as advertised, doing nothing, but doing it very well:

```
>>> do_nothing()
>>>
```

Now let's define and call another function that has no parameters but prints a single word:

```
>>> def make_a_sound():
...     print('quack')
...
>>> make_a_sound()
quack
```

When you call the `make_a_sound()` function, Python runs the code inside its definition. In this case, it prints a single word and returns to the main program.

Let's try a function that has no parameters but *returns* a value:

```
>>> def agree():
...     return True
...
```

You can call this function and test its returned value by using `if`:

```
>>> if agree():
...     print('Splendid!')
... else:
...     print('That was unexpected.')
...
Splendid!
```

You've just taken a big step. Combining functions with tests such as `if` and loops such as `while` makes it easier for you to do tasks that you could not do before.

Arguments and Parameters

At this point, it's time to put something within those parentheses. Let's define the `echo()` function with one parameter called `anything`. It uses the `return` statement to send the value of `anything` back to its caller twice, with a space between:

```
>>> def echo(anything):
...     return anything + ' ' + anything
...
>>>
```

Now let's call `echo()` with the string `Rumplestiltskin`:

```
>>> echo('Rumplestiltskin')
'Rumplestiltskin Rumplestiltskin'
```

The values you pass into the function when you call it are known as *arguments*. When you call a function with arguments, the values of those arguments are copied to their corresponding *parameters* inside the function.

Saying it another way: they're called *arguments* outside the function, but *parameters* inside.

In the previous example, the `echo()` function is called with the argument string `Rumplestiltskin`. This value is copied within `echo()` to the parameter `anything` and then returned (in this case doubled, with a space) to the caller.

These function examples are pretty basic. Let's write a function that takes an input argument and does a bit more with it. We'll adapt the earlier code fragment that comments on a color. Call it `commentary` and have it take an input string parameter called `color`. Make it return the string description to its caller, which can decide what to do with it:

```
>>> def commentary(color):
...     if color == 'red':
...         return "It's a tomato."
...     elif color == "green":
...         return "It's a green pepper."
...     elif color == 'bee purple':
...         return "I don't know what it is, but only bees can see it."
...     else:
...         return "I've never heard of the color "  + color +  "."
...
>>>
```

Call the function `commentary()` with the string argument `blue`:

```
>>> comment = commentary('blue')
```

The function does the following:

- Assigns the value `blue` to the function's internal `color` parameter
- Runs through the `if-elif-else` logic chain
- Returns a string

The caller then assigns the string to the variable `comment`.

What did we get back?

```
>>> print(comment)
I've never heard of the color blue.
```

A function can take any number of input arguments (including zero) of any type. It returns one result, of any type.

A function may appear to return multiple comma-separated values, but it's a single tuple.

If a function doesn't call `return` explicitly, the caller gets the result `None`.

```
>>> print(do_nothing())
None
```

None, Truthiness, and Falsiness

`None` is a special *singleton* Python value that holds a place when there is nothing to say.[1] `None` is not the same as the Boolean value `False`, although it looks false when evaluated as a Boolean. Here's an example:

```
>>> thing = None
>>> bool(thing)
False
>>> if thing:
...     print("It's some thing")
... else:
...     print("It's no thing")
...
It's no thing
```

1 A *singleton* is a unique Python object. There's only one.

To distinguish `None` from a Boolean `False` value, use Python's `is` operator:

```
>>> thing = None
>>> if thing is None:
...     print("It's nothing")
... else:
...     print("It's something")
...
It's nothing
```

You use `is` to compare object *identities*, not their *values*. `True`, `False`, and `None` are all singletons:

```
>>> thing = 0
>>> bool(thing)
False
>>> thing == False
True
>>> thing is False
False
```

```
>>> thing = 1
>>> bool(thing)
True
>>> thing == True
True
>>> thing is True
False
```

This seems like a subtle distinction, but it's important in Python. You'll need `None` to distinguish a missing value from an empty value. Remember that zero-valued integers or floats, empty strings (`''`), lists (`[]`), tuples (`(,)`), dictionaries (`{}`), and sets (`set()`) all evaluate to `False` but are not the same as `None`.

Let's write a quick function that prints whether its argument is `None`, or *evaluates as* `true` or `false`:

```
>>> def whatis(thing):
...     if thing is None:
...         print("None")
...     elif thing:
...         print("true")
...     else:
...         print("false")
...
```

Let's run some sanity tests:

```
>>> whatis(None)
None
>>> whatis(True)
true
```

```
>>> whatis(False)
false
```

How about some real values?

```
>>> whatis(0)
false
>>> whatis(0.0)
false
>>> whatis('')
false
>>> whatis("")
false
>>> whatis('''''')
false
>>> whatis(())
false
>>> whatis([])
false
>>> whatis({})
false
>>> whatis(set())
false
>>> whatis(0.00001)
true
>>> whatis([0])
true
>>> whatis([''])
true
>>> whatis(' ')
true
```

Positional Arguments

Python handles function arguments in a manner that's very flexible when compared to many languages. The most familiar types of arguments are *positional arguments*, whose values are copied to their corresponding parameters in order.

This function builds a dictionary from its positional input arguments and returns it:

```
>>> def menu(wine, entree, dessert):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
...
>>> menu('chardonnay', 'chicken', 'cake')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
```

Although common, a downside of positional arguments is that you need to remember the meaning of each position. If we forget and call `menu()` with wine as the last argument instead of the first, the meal will be very different:

```
>>> menu('beef', 'bagel', 'bordeaux')
{'wine': 'beef', 'entree': 'bagel', 'dessert': 'bordeaux'}
```

Keyword Arguments

To avoid positional argument confusion, you can specify arguments by the names of their corresponding parameters, even in a different order from their definition in the function:

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'wine': 'bordeaux', 'entree': 'beef', 'dessert': 'bagel'}
```

You can mix positional and keyword arguments. Let's specify the wine first, but use keyword arguments for the entree and dessert:

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'wine': 'frontenac', 'entree': 'fish', 'dessert': 'flan'}
```

If you call a function with both positional and keyword arguments, the positional arguments need to come first.

Default Parameter Values

You can specify default values for parameters. The default is used if the caller does not provide a corresponding argument. This bland-sounding feature can be quite useful. Let's use the previous example:

```
>>> def menu(wine, entree, dessert='pudding'):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

This time, try calling `menu()` without the `dessert` argument:

```
>>> menu('chardonnay', 'chicken')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'pudding'}
```

If you do provide an argument, it's used instead of the default:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'wine': 'dunkelfelder', 'entree': 'duck', 'dessert': 'doughnut'}
```

Default parameter values are calculated when the function is *defined*, not when it is run. A common error with new (and sometimes not so new) Python programmers is to use a mutable data type such as a list or dictionary as a default parameter.

In the following test, the `buggy()` function is expected to run each time with a fresh empty `result` list, add the `arg` argument to it, and then print a single-item list. However, there's a bug: the list is empty only the first time it's called. The second time, `result` still has one item from the previous call:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
```

```
...
>>> buggy('a')
['a']
>>> buggy('b')   # expect ['b']
['a', 'b']
```

It would have worked if it had been written like this:

```
>>> def works(arg):
...     result = []
...     result.append(arg)
...     return result
...
>>> works('a')
['a']
>>> works('b')
['b']
```

The fix is to pass in something else to indicate the first call:

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...
>>> nonbuggy('a')
['a']
>>> nonbuggy('b')
['b']
```

This is sometimes a Python job interview question. You've been warned.

Pack/Unpack Positional Arguments with *

If you've programmed in C or C++, you might assume that an asterisk (`*`) in a Python program has something to do with a *pointer*. Technically, Python lets you create C-style pointers with the `ctypes` module, but you would never use them unless you were integrating Python and C code.

When used inside the function with a parameter, an asterisk groups a variable number of positional arguments into a single tuple of parameter values. In the following example, `args` is the parameter tuple that resulted from zero or more arguments that were passed to the function `print_args()`:

```
>>> def print_args(*args):
...     print('Positional tuple:', args)
...
```

If you call the function with no arguments, you get nothing in `*args`:

```
>>> print_args()
Positional tuple: ()
```

Whatever you give it will be printed as the `args` tuple:

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')
Positional tuple: (3, 2, 1, 'wait!', 'uh...')
```

This is useful for writing functions such as `print()` that accept a variable number of arguments. If your function has *required* positional arguments as well, put them first; `*args` goes at the end and grabs all the rest:

```
>>> def print_more(required1, required2, *args):
...     print('Need this one:', required1)
...     print('Need this one too:', required2)
...     print('All the rest:', args)
...
>>> print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')
Need this one: cap
Need this one too: gloves
All the rest: ('scarf', 'monocle', 'mustache wax')
```

When using *, you don't need to name the tuple parameter `*args`, but it's a standard idiom in Python.

Let's summarize:

- You can pass positional arguments to a function, which will match them inside to positional parameters. This is what you've seen so far in this book.
- You can pass a tuple argument to a function, and inside it will be a tuple parameter. This is a simple case of the preceding one.
- You can pass positional arguments to a function, and gather them inside as the parameter `*args`, which resolves to the tuple `args`. This was described in this section.
- You can also "explode" a tuple argument called `args` to positional parameters `*args` inside the function, which will be regathered inside into the tuple parameter `args`:

```
>>> print_args(2, 5, 7, 'x')
Positional tuple: (2, 5, 7, 'x')
>>> args = (2,5,7,'x')
>>> print_args(args)
Positional tuple: ((2, 5, 7, 'x'),)
>>> print_args(*args)
Positional tuple: (2, 5, 7, 'x')
```

You can only use the * syntax in a function call or definition:

```
>>> *args
  File "<stdin>", line 1
SyntaxError: can't use starred expression here
```

Let's summarize:

- Outside the function, `*args` explodes the tuple `args` into comma-separated positional parameters.
- Inside the function, `*args` gathers all the positional arguments into a single `args` tuple. You could use the names `*params` and `params`, but using `*args` for both the outside argument and inside parameter is common practice.

Readers with synesthesia might also faintly hear `*args` as *puff-args* on the outside and *inhale-args* on the inside, as values are either exploded or gathered.

Pack/Unpack Keyword Arguments with **

You can use two asterisks (**) to group keyword arguments into a dictionary, where the argument names are the keys, and their values are the corresponding dictionary values. The following example defines the `print_kwargs()` function to print its keyword arguments:

```
>>> def print_kwargs(**kwargs):
...     print('Keyword arguments:', kwargs)
...
```

Using the variable name `kwargs` (meaning *keyword arguments*) with the ** is traditional but not required.

Now try calling `print_kwargs()` with some keyword arguments:

```
>>> print_kwargs()
Keyword arguments: {}
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot',
'entree': 'mutton'}
```

Inside the function, `kwargs` is a dictionary parameter.

You'll generally see this argument order:

1. Required positional arguments
2. Optional positional arguments (`*args`)
3. Optional keyword arguments (`**kwargs`)

See the next section for details on how to specify *keyword-only* and *position-only* arguments. See the Python documentation for the details.

As with `args`, you don't need to call this keyword argument `kwargs`, but that usage is common.[2]

The `**` syntax is valid only in a function call or definition:[3]

```
>>> **kwparams
  File "<stdin>", line 1
    **kwparams
     ^
SyntaxError: invalid syntax
```

Let's summarize:

- You can pass keyword arguments to a function, which will match them inside to keyword parameters. This is what you've seen so far.
- You can pass a dictionary argument to a function, and inside it will be dictionary parameters. This is a simple case of the preceding one.
- You can pass one or more keyword arguments (*`name=value`*) to a function, and gather them inside as `**kwargs`, which resolves to the dictionary parameter called `kwargs`. This was described in this section.
- Outside a function, `**kwargs` *explodes* a dictionary `kwargs` into *`name=value`* arguments.
- Inside a function's parameter section, `**kwargs` *gathers* *`name=value`* arguments into the single dictionary parameter `kwargs`.

If auditory hallucinations help, imagine a puff for each asterisk exploding outside the function, and a little inhaling sound for each one gathering inside.

Keyword-Only (*) and Position-Only (/) Arguments

You may want an argument to be in a defined *position*, and that requires the use of a single slash (`/`) just after the position-only parameters:

```
>>> def print_data(data, /, start=0, end=100):
...     for value in (data[start:end]):
```

2 Although *Args* and *Kwargs* sound like the names of pirate parrots.

3 Or, as of Python 3.5, the `**` syntax is valid in a dictionary merge of the form `{**a, **b}`, as you saw in Chapter 5.

```
...         print(value)
...
>>> print_data(data, 2, 4)
c
d
>>> print_data(data, end=5, start=3)
d
e
```

It's possible to pass in a keyword argument that has the same name as a positional parameter, probably not resulting in what you want. Python lets you specify *keyword-only arguments*. As the name says, they must be provided as *`name=value`*, not positionally as *`value`*.

First, let's define a function that takes an iterable `data` argument and optional `start` and `end` indices:

```
>>> def print_data(data, start=0, end=100):
...     for value in (data[start:end]):
...         print(value)
...
>>> data = ['a', 'b', 'c', 'd', 'e', 'f']
```

We could call the arguments out of order by passing them all as keyword arguments:

```
>>> print_data(start=0, end=4, data=data)
a
b
c
d
```

Now let's redefine the `print_data()` function. Adding a `*` in the function definition means that all the following parameters (in this example, `start` and `end`) *must* be provided as named arguments if we don't want their default values:

```
>>> def print_data(data, *, start=0, end=100):
...     for value in (data[start:end]):
...         print(value)
...
>>> data = ['a', 'b', 'c', 'd', 'e', 'f']
>>> print_data(data, start=0, end=4)
a
b
c
d
>>> print_data(data, start=4)
e
f
>>> print_data(data, end=2)
a
b
```

If we use both * and /, then data must be the first argument, and start and end must be specified by keyword:

```
>>> def print_data(data, /, *, start=0, end=100):
...     for value in (data[start:end]):
...         print(value)
...
>>> print_data(data, 2, 4)
Traceback (most recent call last):
  File "<python-input-37>", line 1, in <module>
    print_data(data, 2, 4)
    ~~~~~~~~~~^^^^^^^^^^^^
TypeError: print_data() takes 1 positional argument but 3 were given
>>> print_data(start=0, end=1, data)
  File "<python-input-38>", line 1
    print_data(start=0, end=1, data)
                                  ^
SyntaxError: positional argument follows keyword argument
```

To recap: use / *after* position-only arguments, and * *before* keyword-only arguments.

Mutable and Immutable Arguments

Remember that if you assign the same list to two variables, you can change it by using either one? And that you cannot do that if the variables both refer to something like an integer or a string? That's because a list is mutable (its value can change), and the integer and string are immutable (hands off).

You need to watch for the same behavior when passing arguments to functions. If an argument is mutable, its value can be changed *from inside the function* via its corresponding parameter:[4]

```
>>> outside = ['one', 'fine', 'day']
>>> def mangle(arg):
...     arg[1] = 'terrible!'
...
>>> outside
['one', 'fine', 'day']
>>> mangle(outside)
>>> outside
['one', 'terrible!', 'day']
```

So, don't do this. Either document that an argument may be changed or return the new value.

4 Like the teens-in-peril movies where they learn "The call's coming from inside the house!"

Docstrings

Readability counts, the Zen of Python verily saith. You can attach documentation to a function definition by including a string at the beginning of the function body. This is the function's *docstring*:

```
>>> def echo(anything):
...     'echo returns its input argument'
...     return anything
```

You can make a docstring quite long, and even add rich formatting if you want:

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
```

To print a function's docstring, call the Python `help()` function. Pass the function's name to get a listing of arguments along with the nicely formatted docstring:

```
>>> help(echo)
Help on function echo in module __main__:

echo(anything)
    echo returns its input argument
```

If you want to see just the raw docstring, without the formatting, use this:

```
>>> print(echo.__doc__)
echo returns its input argument
```

That odd-looking `__doc__` is the internal name of the docstring as a variable within the function. Double underscores (aka *dunder* in Python-speak) are used in many places to name Python internal variables, because programmers are unlikely to use them in their own variable names.

Functions Are First-Class Citizens

In Python, everything that has a value is an object. This includes numbers, strings, tuples, lists, dictionaries—and functions as well. Functions are first-class citizens in Python. You can assign them to variables, use them as arguments to other functions, and return them from functions. This gives you the capability to do some tasks in Python that are difficult-to-impossible to carry out in some other languages.

To test this, let's define a simple function called `answer()` that doesn't have any arguments; it just prints the number `42`:

```
>>> def answer():
...     print(42)
```

If you run this function, you know what you'll get:

```
>>> answer()
42
```

Now let's define another function named `run_something`. It has one argument called `func`, a function to run. Once inside, it just calls the function:

```
>>> def run_something(func):
...     func()
```

If we pass `answer` to `run_something()`, we're using a function as data, just as with anything else:

```
>>> run_something(answer)
42
```

Notice that you passed `answer`, not `answer()`. In Python, those parentheses mean *call this function*. With no parentheses, Python just treats the function like any other object. That's because, like everything else in Python, a function *is* an object:

```
>>> type(run_something)
<class 'function'>
```

Let's try running a function with arguments. Define a function `add_args()` that prints the sum of its two numeric arguments, `arg1` and `arg2`:

```
>>> def add_args(arg1, arg2):
...     print(arg1 + arg2)
```

And what is `add_args()`?

```
>>> type(add_args)
<class 'function'>
```

At this point, let's define a function called `run_something_with_args()` that takes three arguments:

`func`
: The function to run

`arg1`
: The first argument for `func`

`arg2`
: The second argument for `func`

Here's the code:

```
>>> def run_something_with_args(func, arg1, arg2):
...     func(arg1, arg2)
```

When you call `run_something_with_args()`, the function passed by the caller is assigned to the `func` parameter, whereas `arg1` and `arg2` get the values that follow in the argument list. Then, running `func(arg1, arg2)` executes that function with those arguments because the parentheses told Python to do so.

Let's test this function by passing the function name `add_args` and the arguments `5` and `9` to `run_something_with_args()`:

```
>>> run_something_with_args(add_args, 5, 9)
14
```

Within `run_something_with_args()`, the function name argument `add_args` is assigned to the parameter `func`, `5` to `arg1`, and `9` to `arg2`. This ends up running the following:

```
add_args(5, 9)
```

You can combine this with the `*args` and `**kwargs` techniques.

Let's define a test function that takes any number of positional arguments, calculates their sum by using the `sum()` function, and then returns that sum:

```
>>> def sum_args(*args):
...     return sum(args)
```

I haven't mentioned `sum()` before. This built-in Python function calculates the sum of the values in its iterable numeric (int or float) argument.

Let's define the new function `run_with_positional_args()`, which takes a function and any number of positional arguments to pass to it:

```
>>> def run_with_positional_args(func, *args):
...     return func(*args)
```

Now go ahead and call it:

```
>>> run_with_positional_args(sum_args, 1, 2, 3, 4)
10
```

You can use functions as elements of lists, tuples, sets, and dictionaries. Functions are immutable, so you can also use them as dictionary keys.

Function Arguments Are Not a Tuple

Even though we call functions with comma-separated arguments, they're not the same as a tuple:

```
>>> t = 1, 2, 3
>>> type(t)
<class 'tuple'>
>>> def list_arg(thing):
...     print(thing, type(thing))
...
>>> list_arg(t)
(1, 2, 3) <class 'tuple'>
>>> list_arg(1, 2, 3)
Traceback (most recent call last):
  File "<python-input-68>", line 1, in <module>
    list_arg(1, 2, 3)
    ~~~~~~~~^^^^^^^^^
TypeError: list_arg() takes 1 positional argument but 3 were given
>>>
```

If you wrap the arguments in another pair of parentheses, then you have a single tuple:

```
>>> list_arg( (1, 2, 3) )
(1, 2, 3) <class 'tuple'>
```

As you saw in "Pack/Unpack Positional Arguments with *" on page 156, you can implode any number of arguments into a tuple parameter with `*args`:

```
>>> t = 1, 2, 3
>>> def list_args(*args):
...     print(args, type(args))
...
>>> list_args(t)
((1, 2, 3),) <class 'tuple'>
>>> list_args(1, 2, 3)
(1, 2, 3) <class 'tuple'>
```

Inner Functions

You can define a function within another function:

```
>>> def outer(a, b):
...     def inner(c, d):
...         return c + d
...     return inner(a, b)
...
>>>
>>> outer(4, 7)
11
```

An inner function can be useful when performing a complex task more than once within another function, to avoid loops or code duplication. For a string example, this inner function adds text to its argument:

```
>>> def knights(saying):
...     def inner(quote):
...         return f"We are the knights who say: '{quote}'"
...     return inner(saying)
...
>>> knights('Ni!')
"We are the knights who say: 'Ni!'"
```

Closures

An inner function can act as a *closure*. This is a function that is dynamically generated by another function and can both change and remember the values of variables that were created outside the function.

The following example builds on the previous `knights()` example. Let's call the new one `knights2()`, because we have no imagination, and turn the `inner()` function into a closure called `inner2()`. Here are the differences:

- `inner2()` uses the outer `saying` parameter directly instead of getting it as an argument.
- `knights2()` returns the `inner2` function name instead of calling it:

```
>>> def knights2(saying):
...     def inner2():
...         return "We are the knights who say: '%s'" % saying
...     return inner2
...
```

The `inner2()` function knows the value of `saying` that was passed in and remembers it. The line `return inner2` returns this specialized copy of the `inner2` function (but doesn't call it). That's a kind of closure: a dynamically created function that remembers where it came from.

Let's call `knights2()` twice, with different arguments:

```
>>> a = knights2('Duck')
>>> b = knights2('Hasenpfeffer')
```

OK, so what are `a` and `b`?

```
>>> type(a)
<class 'function'>
>>> type(b)
<class 'function'>
```

They're functions, but they're also closures:

```
>>> a
<function knights2.<locals>.inner2 at 0x10193e158>
>>> b
<function knights2.<locals>.inner2 at 0x10193e1e0>
```

If we call them, they remember the `saying` that was used when they were created by `knights2`:

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

Anonymous Functions: Lambda

A Python *lambda function* is an anonymous function expressed as a single statement. You can use it instead of a normal tiny function.

To illustrate a lambda function, let's first make an example that uses normal functions. To begin, let's define the function `edit_story()`. Its arguments are the following:

`words`
: A list of words

`func`
: A function to apply to each word in `words`

Here's the function:

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

Now we need a list of words and a function to apply to each word. For the words, here's a list of (hypothetical) sounds made by my cat if she (hypothetically) missed one of the stairs:

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

And for the function, let's use one that will capitalize each word and append an exclamation point, perfect for feline tabloid newspaper headlines:

```
>>> def enliven(word):   # give that prose more punch
...     return word.capitalize() + '!'
```

Then we mix our ingredients:

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Finally, we get to the lambda. The `enliven()` function is so brief that we can replace it with a lambda:

```
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
```

A lambda has zero or more comma-separated arguments, followed by a colon (`:`), and then the definition of the function. We're giving this lambda one argument, `word`. You don't use parentheses with `lambda` as you would when calling a function created with `def`.

Often, using real functions such as `enliven()` is much clearer than using lambdas. Lambdas are mostly useful when you would otherwise need to define many tiny functions and remember what you called them all.

Generators

A *generator* is a Python sequence creation object. With it, you can iterate through potentially huge sequences without creating and storing the entire sequence in memory at once. Generators are often the source of data for iterators. As you may recall, we already used one of them, `range()`, in earlier code examples to generate a series of integers. which became the normal `range()` in Python 3. This example adds all the integers from 1 to 100:

```
>>> sum(range(1, 101))
5050
```

Every time you iterate through a generator, it keeps track of where it was the last time it was called and returns the next value. This is different from a normal function, which has no memory of previous calls and always starts at its first line with the same state.

Generator Functions

If you want to create a potentially large sequence, you can write a *generator function*. This function returns its value with a `yield` statement rather than `return`. Let's write our own simplified (positive steps only) version of `range()`:

```
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
```

It's a function:

```
>>> my_range
<function my_range at 0x10193e268>
```

And it returns a generator object:

```
>>> ranger = my_range(1, 5)
>>> ranger
<generator object my_range at 0x101a0a168>
```

We can iterate over this returned generator object:

```
>>> for x in ranger:
...     print(x)
...
1
2
3
4
```

A simple generator can be run only once. Lists, sets, strings, and dictionaries exist in memory, but a generator creates its values on the fly and hands them out one at a time through an iterator. It doesn't remember them, so you can't restart or back up a generator.

If you try to iterate this generator again, you'll find that it's tapped out:

```
>>> for try_again in ranger:
...     print(try_again)
...
>>>
```

Still, there are ways of making a generator function restartable:

- Return a new generator. In the preceding examples, call `ranger = my_range(1, 5)` again, then iterate over this new `ranger` generator object.
- Define a generator object in a nested function:

```
>>> def my_range(first=0, last=10, step=1):
...     def inner():
...         number = first
...         while number < last:
...             yield number
...             number += step
...     return inner
...
```

```
... ranger = my_range(1, 5)
... for x in ranger():
...     print(x)
... for x in ranger():
...     print(x)
...
1
2
3
4
1
2
3
4
```

- Define an object class to wrap the generator code (the `yield` parts). I'll skip the details on this for now, because we won't get to classes until Chapter 11.

Generator Comprehensions

You've seen comprehensions for lists, dictionaries, and sets. A *generator comprehension* looks like those but is surrounded by parentheses instead of square brackets or curly braces. A generator comprehension is like a shorthand version of a generator function, doing the `yield` invisibly, and also returns a generator object:

```
>>> genobj = (pair for pair in zip(['a', 'b'], ['1', '2']))
>>> genobj
<generator object <genexpr> at 0x10308fde0>
>>> for thing in genobj:
...     print(thing)
...
('a', '1')
('b', '2')
```

Decorators

Sometimes you want to modify an existing function without changing its source code. A common example is adding a debugging statement to see which arguments were passed in.

A *decorator* is a function that takes one function as input and returns another function. Let's dig into our bag of Python tricks and use the following:

- `*args` and `**kwargs`
- Inner functions
- Functions as arguments

The `document_it()` function defines a decorator that will do the following:

- Print the function's name and the values of its arguments
- Run the function with the arguments
- Print the result
- Return the modified function for use

Here's what the code looks like:

```
>>> def document_it(func):
...     def new_function(*args, **kwargs):
...         print('Running function:', func.__name__)
...         print('Positional arguments:', args)
...         print('Keyword arguments:', kwargs)
...         result = func(*args, **kwargs)
...         print('Result:', result)
...         return result
...     return new_function
```

Whatever `func` you pass to `document_it()`, you get a new function that includes the extra statements that `document_it()` adds. A decorator doesn't have to run any code from `func`, but `document_it()` calls `func` partway through so that you get the results of `func` as well as all the extras.

How do you use this? You can apply the decorator manually:

```
>>> def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
8
>>> cooler_add_ints = document_it(add_ints)  # manual decorator assignment
>>> cooler_add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

As an alternative to the manual decorator assignment we just looked at, you can add *`@decorator_name`* before the function that you want to decorate:

```
>>> @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
```

```
Result: 8
8
```

You can have more than one decorator for a function. Let's write another decorator called `square_it()` that squares the result:

```
>>> def square_it(func):
...     def new_function(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result * result
...     return new_function
...
```

The decorator that's used closest to the function (just above the `def`) runs first and then the one above it. Either order gives the same end result, but you can see how the intermediate steps change:

```
>>> @document_it
... @square_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: new_function
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 64
64
```

Let's try reversing the decorator order:

```
>>> @square_it
... @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
64
```

One last suggestion about decorators: use the `wraps` decorator from the `functools` module. This copies some extra data (like the function name and any docstring) from the original function to its decorated version. Here's where you'd put it:

```
>>> from functools import wraps
>>> def outer():
...     @wraps(func)
...     def inner(*args, **kwargs):
...         pass
...     return inner
```

Namespaces and Scope

Desiring this man's art and that man's scope
—William Shakespeare

A name can refer to different things, depending on where it's used. Python programs have various *namespaces*—sections within which a particular name is unique and unrelated to the same name in other namespaces.

Each function defines its own namespace. If you define a variable called `x` in a main program and another variable called `x` in a function, they refer to different things. But the walls can be breached: if you need to, you can access names in other namespaces in various ways.

The main part of a program defines the *global* namespace; thus, the variables in that namespace are *global variables*. You can get the value of a global variable from within a function:

```
>>> animal = 'fruitbat'
>>> def print_global():
...     print('inside print_global:', animal)
...
>>> print('at the top level:', animal)
at the top level: fruitbat
>>> print_global()
inside print_global: fruitbat
```

But if you try to *change* the value of the global variable within the function, you get an error:

```
>>> def change_and_print_global():
...     print('inside change_and_print_global:', animal)
...     animal = 'wombat'
...     print('after the change:', animal)
...
>>> change_and_print_global()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in change_and_print_global
UnboundLocalError: local variable 'animal' referenced before assignment
```

If you just assign a value to the variable `animal`, it creates a variable of that name, but only inside the function:

```
>>> def change_local():
...     animal = 'wombat'
...     print('inside change_local:', animal, id(animal))
...
>>> change_local()
inside change_local: wombat 4330406160
>>> animal
'fruitbat'
>>> id(animal)
4330390832
```

What happened here? The first line assigns the string `fruitbat` to a global variable named `animal`. The `change_local()` function also has a variable named `animal`, but that's in its local namespace.

I used the Python function `id()` here to print the unique value for each object and prove that the variable `animal` inside `change_local()` is not the same as `animal` at the main level of the program.

To access the global variable rather than the local one within a function, you need to be explicit and use the `global` keyword (you knew this was coming: *explicit is better than implicit*):

```
>>> animal = 'fruitbat'
>>> def change_and_print_global():
...     global animal
...     animal = 'wombat'
...     print('inside change_and_print_global:', animal)
...
>>> animal
'fruitbat'
>>> change_and_print_global()
inside change_and_print_global: wombat
>>> animal
'wombat'
```

If you don't say `global` within a function, Python uses the local namespace, and the variable is local if it's defined there. The variable goes away after the function completes.

Python provides two functions to access the contents of your namespaces:

- `locals()` returns a dictionary of the contents of the local namespace.
- `globals()` returns a dictionary of the contents of the global namespace.

And here they are in use:

```
>>> animal = 'fruitbat'
>>> def change_local():
...     animal = 'wombat'  # local variable
...     print('locals:', locals())
...
>>> animal
'fruitbat'
>>> change_local()
locals: {'animal': 'wombat'}
>>> print('globals:', globals()) # reformatted a little for presentation
globals: {'animal': 'fruitbat',
'__doc__': None,
'change_local': <function change_local at 0x1006c0170>,
'__package__': None,
```

```
'__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__builtins__': <module 'builtins'>}
>>> animal
'fruitbat'
```

The local namespace within `change_local()` contains only the local variable `animal`. The global namespace contains the separate global variable `animal` and a number of other elements.

Dunder Names

Names that begin and end with double underscores (`__`, nicknamed *dunder*) are reserved for use within Python, so you should not use them with your own variables. This naming pattern was chosen because it seemed unlikely to be selected by application developers for their own variables.

For instance, the name of a function is in the system variable *`function`*`.__name__`, and its documentation string is *`function`*`.__doc__`:

```
>>> def amazing():
...     '''This is the amazing function.
...     Want to see it again?'''
...     print('This function is named:', amazing.__name__)
...     print('And its docstring is:', amazing.__doc__)
...
>>> amazing()
This function is named: amazing
And its docstring is: This is the amazing function.
    Want to see it again?
```

As you saw in the earlier `globals` printout, the main program is assigned the special name `'__main__'`.

Recursion

So far, we've called functions that take some actions directly, and maybe call other functions. But what if a function calls itself?[5] This is called *recursion*. Like an unbroken infinite loop with `while` or `for`, you don't want infinite recursion. Do we still need to worry about cracks in the space-time continuum? It's a wonder it's lasted this long.

5 It's like saying, "I wish I had a dollar for every time I wished I had a dollar."

Python saves the universe again by raising an exception if you get too deep:

```
>>> def dive():
...     return dive()
...
>>> dive()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in dive
  File "<stdin>", line 2, in dive
  File "<stdin>", line 2, in dive
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Recursion is useful when you're dealing with "uneven" data, like lists of lists of lists. Suppose that you want to "flatten" all sublists of a list, no matter how deeply nested.[6] A generator function is just the thing:

```
>>> def flatten(lol):
...     for item in lol:
...         if isinstance(item, list):
...             for subitem in flatten(item):
...                 yield subitem
...         else:
...             yield item
...
>>> lol = [1, 2, [3,4,5], [6,[7,8,9], []]]
>>> flatten(lol)
<generator object flatten at 0x10509a750>
>>> list(flatten(lol))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python 3.3 added the `yield from` expression, which lets a generator hand off some work to another generator. We can use it to simplify `flatten()`:

```
>>> def flatten(lol):
...     for item in lol:
...         if isinstance(item, list):
...             yield from flatten(item)
...         else:
...             yield item
...
>>> lol = [1, 2, [3,4,5], [6,[7,8,9], []]]
>>> list(flatten(lol))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6 Another Python interview question. Collect the whole set!

Async Functions, Briefly

For most of Python's existence, code was executed a line at a time, in order. If you called something that took a while, like accessing a web service or reading a file, you couldn't do anything else until it finished.

Well, sort of. More recently, the keywords `async` and `await` were added to Python 3.5 to define and run *asynchronous functions*.

You'll recognize an asynchronous function in two ways:

- It starts with `async def` instead of just `def`.
- It may contain at least one line containing the keyword `await`. A normal function (defined with `def`) cannot contain an `await`.

For multiple reasons, I've pushed the details on all things async to Chapter 23, which covers the overall issue of *concurrency*, because of the following:

- Async is confusing.
- You don't actually need to ever write an async function, although you should understand how one works.
- Alternative techniques, such as threads and multiprocessing, are available.

Exceptions

In some languages, errors are indicated by special function return values. When things go south,[7] Python uses *exceptions*: code that is executed when an associated error occurs.

You've seen some of these already, such as accessing a list or tuple with an out-of-range position, or a dictionary with a nonexistent key. When you run code that might fail under some circumstances, you also need appropriate *exception handlers* to intercept any potential errors.

Adding exception handling anywhere an exception might occur is a good practice, to let the user know what is happening. You might not be able to fix the problem, but at least you can note the circumstances and shut down your program gracefully. If an exception occurs in a function and is not caught there, it *bubbles up* until it is caught by a matching handler in a calling function. If you don't provide your own exception handler, Python prints an error message and information about where the error occurred and then terminates the program, as demonstrated in the following snippet:

7 Is this Northern Hemispherism? Do Aussies and Kiwis say that things go "north" when they mess up?

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Handle Errors with try and except

> *Do, or do not. There is no try.*
>
> —Yoda

Rather than leaving success to chance, use `try` to wrap your code, and `except` to provide the error handling:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> try:
...     short_list[position]
... except:
...     print('Need a position between 0 and', len(short_list)-1, ' but got',
...           position)
...
Need a position between 0 and 2 but got 5
```

The code inside the `try` block is run. If an error occurs, an exception is raised and the code inside the `except` block runs. If no errors occur, the `except` block is skipped.

Specifying a plain `except` with no arguments, as we did here, is a catchall for any exception type. If more than one type of exception could occur, providing a separate exception handler for each is best. No one forces you to do this; you can use a bare `except` to catch all exceptions, but your treatment of them would probably be generic (something akin to printing *Some error occurred*). You can use any number of specific exception handlers.

Sometimes you want exception details beyond the type. You get the full exception object in the variable *name* if you use this form:

```
except exceptiontype as name
```

The example that follows looks for an `IndexError` first, because that's the exception type raised when you provide an illegal position to a sequence. This code saves an `IndexError` exception in the variable `err`, and any other exception in the variable `other`. The example prints everything stored in `other` to show what you get in that object:

```
>>> short_list = [1, 2, 3]
>>> while True:
...     value = input('Position [q to quit]? ')
...     if value == 'q':
...         break
...     try:
...         position = int(value)
...         print(short_list[position])
...     except IndexError:
...         print('Bad index:', position)
...     except Exception as other:
...         print('Something else broke:', other)
...
Position [q to quit]? 1
2
Position [q to quit]? 0
1
Position [q to quit]? 2
3
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? 2
3
Position [q to quit]? two
Something else broke: invalid literal for int() with base 10: 'two'
Position [q to quit]? q
```

Inputting position `3` raises an `IndexError` as expected. Entering `two` annoys the `int()` function, which we handle in our second catchall `except` code.

Use Finally

The try/except block may have a closing `finally` section. If it exists, it is *always* called, whether the `try` part or any `except` part succeeded. The `finally` section is meant to contain code to handle the result regardless of whether the previous code had a happy ending. An example might be to close an open file or make a final remark:

```
>>> try:
...     100 / 0
...     print("I don't know how I did this.")
... except ZeroDivisionError:
...     print("Really?")
... finally:
...     print("Can life go on now?")
...
Really?
Can life go on now?
>>>
```

Make Your Own Exceptions

The previous sections discussed handling exceptions, but all the exceptions (such as `IndexError`) were predefined in Python or its standard library. You can use any of these for your own purposes. You can also define your own exception types to handle special situations that might arise in your own programs.

Creating a new exception requires defining a new object type with a *class*—something we don't get into until Chapter 11. So, if you're unfamiliar with classes, you might want to return to this section later.

An exception is a class. It is a child of the class `Exception`. Let's make an exception called `UppercaseException` and raise it when we encounter an uppercase word in a string:

```
>>> class UppercaseException(Exception):
...     pass
...
>>> words = ['eenie', 'meenie', 'miny', 'MO']
>>> for word in words:
...     if word.isupper():
...         raise UppercaseException(word)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
__main__.UppercaseException: MO
```

We don't even define any behavior for `UppercaseException` (notice we used just `pass`), letting its parent class `Exception` figure out what to print when the exception is raised.

You can access the exception object itself and print it:

```
>>> try:
...     raise OopsException('panic')
... except OopsException as exc:
...     print(exc)
...
panic
```

Review/Preview

We're starting to get serious about programming now, with functions. They're the first big step into large-scale code structures, essential for writing any serious programs.

The next chapter is a deep dive into objects: a core concept for an object-oriented language like Python. This is often a confusing topic, sometimes with more opinions than practical advice. Luckily, there will be much more of the latter.

Practice

10.1 Define a function called `good()` that returns the following list: `['Harry', 'Ron', 'Hermione']`.

10.2 Define a generator function called `get_odds()` that returns the odd numbers from `range(10)`. Use a `for` loop to find and print the third value returned.

10.3 Define a decorator called `test` that prints `start` when a function is called, and `end` when it finishes.

10.4 Define an exception called `OopsException`. Raise this exception to see what happens. Then write the code to catch this exception and print `Caught an oops`.

CHAPTER 11
Objects

No object is mysterious. The mystery is your eye.
—Elizabeth Bowen

Take an object. Do something to it. Do something else to it.
—Jasper Johns

As I've mentioned on various pages, everything in Python, from numbers to functions, is an object. However, Python hides most of the object machinery by means of special syntax. You can type `num = 7` to create an object of type integer with the value 7, and assign an object reference to the name `num`. The only time you need to look inside objects is when you want to make your own or modify the behavior of existing objects. You'll see how to do both in this chapter.

What Are Objects?

An *object* is a data structure containing both data (variables, called *attributes*) and code (functions, called *methods*). An object represents a unique instance of a concrete thing. Think of objects as nouns and their methods as verbs. An object represents an individual thing, and its methods define the way it interacts with other things.

For example, the integer object with the value `7` is an object that facilitates methods such as addition and multiplication, as you saw in Chapter 3. But `8` is a different object. This means an integer class is built in somewhere in Python, to which both `7` and `8` belong. The strings `cat` and `duck` are also objects in Python, and have string methods that you've seen in Chapter 4, such as `capitalize()` and `replace()`.

Unlike modules (coming in Chapter 12), you can have multiple objects of the same class (often referred to as *instances*) at the same time, each with potentially different attributes. They're like super data structures, with code thrown in.

Simple Objects

Let's start with basic object classes; we'll save the discussion of *inheritance* for a few pages.

Define a Class with class

To create a new object that no one has ever created before, you first define a *class* that indicates what that object contains.

In Chapter 2, I compared an object to a plastic box. A *class* is like the mold that makes that box. For instance, Python has a built-in class that makes string objects such as `cat` and `duck`, and the other standard data types—lists, dictionaries, and so on. To create your own custom object in Python, you first need to define a class by using the `class` keyword. Let's walk through some simple examples.

Suppose that you want to define objects to represent information about cats.[1] Each object will represent one feline. You'll first want to define a class called `Cat` as the mold. In the examples that follow, we try more than one version of this class as we build up from the simplest class to ones that do something useful.

We're following the naming conventions of Python's PEP-8.

Our first try is the simplest possible class, an empty one:

```
>>> class Cat():
...     pass
```

You can also define the class without parentheses:

```
>>> class Cat:
...     pass
```

Just as with functions, we need to say `pass` to indicate that this class is empty. This definition is the bare minimum to create an object.

1 Or even if you don't want to.

You create an object from a class by calling the class name as though it were a function:

```
>>> a_cat = Cat()
>>> another_cat = Cat()
```

In this case, calling `Cat()` creates two individual objects from the `Cat` class, and we assign them to the names `a_cat` and `another_cat`. But our `Cat` class has no other code, so the objects that we create from it just sit there and can't do much else.

Well, they can do a little.

Assign Attributes

An *attribute* is a variable inside a class or object. When an object or class is created, as well as afterward, you can assign attributes to it. An attribute can be any other object. Let's make two cat objects again:

```
>>> class Cat:
...     pass
...
>>> a_cat = Cat()
>>> a_cat
<__main__.Cat object at 0x100cd1da0>
>>> another_cat = Cat()
>>> another_cat
<__main__.Cat object at 0x100cd1e48>
```

When we defined the `Cat` class, we didn't specify how to print an object from that class. Python jumps in and prints something like `<__main__.Cat object at 0x100cd1da0>`. In "Magic Methods" on page 205, you'll see how to change this default behavior.

Now assign a few attributes to our first object:

```
>>> a_cat.age = 3
>>> a_cat.name = "Mr. Fuzzybuttons"
>>> a_cat.nemesis = another_cat
```

Can we access these? We sure hope so:

```
>>> a_cat.age
3
>>> a_cat.name
'Mr. Fuzzybuttons'
>>> a_cat.nemesis
<__main__.Cat object at 0x100cd1e48>
```

Because `nemesis` is an attribute referring to another `Cat` object, we can use `a_cat.nemesis` to access it, but this other object doesn't have a `name` attribute yet:

```
>>> a_cat.nemesis.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'name'
```

Let's name our archfeline:

```
>>> a_cat.nemesis.name = "Mr. Bigglesworth"
>>> a_cat.nemesis.name
'Mr. Bigglesworth'
```

Even the simplest object like this one can be used to store multiple attributes. So, you can use multiple objects to store different values, instead of using something like a list or dictionary.

When you hear *attributes*, it usually means object attributes. Python also has *class attributes*, and you'll see the differences in "Class and Object Attributes" on page 200.

Methods

A *method* is a function in a class or object. A method looks like any other function but can be used in special ways that you'll see in "Properties for Attribute Access" on page 197 and "Method Types" on page 201.

Initialization

An object instance is first created with the `__new__()` method, but this happens behind the scenes, and you normally don't even see this. The next step is *initialization*, handled by `__init__()`: Here's a minimal example:

```
>>> class Cat:
...     def __init__(self):
...         pass
```

This is what you'll see in real Python class definitions. I admit that `__init__()` and `self` look strange. That `__init__()` is the special Python name for a method that initializes an individual object from its class definition.[2] The `self` argument specifies that it refers to the individual object itself.

When you define `__init__()` in a class definition, its first parameter should be named `self`. Although `self` is not a reserved word in Python, it's common usage. No one reading your code later (including you!) will need to guess what you meant if you use `self`.

2 As I've mentioned earlier, these double-underscore names are nicknamed *dunder*. You may flash to "Dunder-Mifflin" in the American version of *The Office* if you are a fan.

But even this second `Cat` class definition doesn't create an object that really does anything. The third try is the charm that really shows how to create a simple object in Python and assign one of its attributes. This time, we add the parameter `name` to the initialization method:

```
>>> class Cat():
...     def __init__(self, name):
...         self.name = name
...
>>>
```

Now we can create an object from the `Cat` class by passing a string for the `name` parameter:

```
>>> furball = Cat('Grumpy')
```

Here's what this line of code does:

- Looks up the definition of the `Cat` class
- *Instantiates* (creates) a new object in memory
- Calls the object's `__init__()` method, passing this newly created object as `self` and the other argument (`Grumpy`) as `name`
- Stores the value of `name` in the object
- Returns the new object
- Attaches the variable `furball` to the object

This new object is like any other object in Python. You can use it as an element of a list, tuple, dictionary, or set. You can pass the object to a function as an argument, or return it as a result.

What about the `name` value that we passed in? It is saved with the object as an attribute. You can read and write it directly:

```
>>> print('Our latest addition: ', furball.name)
Our latest addition: Grumpy
```

Remember, *inside* the `Cat` class definition, you access the `name` attribute as `self.name`. When you create an actual object and assign it to a variable like `furball`, you refer to it as `furball.name`.

It is *not* necessary to have an `__init__()` method in every class definition; this method is used to do anything that's needed to distinguish this object from others created from the same class. It's not what some other languages would call a *constructor*. Python already constructed the object for you. Think of `__init__()` as an *initializer*.

You can make many individual objects from a single class. But remember that Python implements data as objects, so the class itself is also an object. However, your program has only one class object. If you defined `class Cat` as we did here, it's like the Highlander—there can be only one—in that namespace.

Inheritance

When you're trying to solve a coding problem, often you'll find an existing class that creates objects that do almost what you need. What can you do?

You could modify this old class, but you'll make it more complicated, and you might break something that used to work. Or you could write a new class, cutting and pasting from the old one and merging your new code. But this means that you have more code to maintain, and the parts of the old and new classes that used to work the same might drift apart because they're now in separate places.

One solution is *inheritance*: creating a new class *from* an existing class, but with additions or changes. This approach is a good way to reuse code. When you use inheritance, the new class can automatically use all the code from the old class but without you needing to copy any of it.

The current consensus favors *composition* over inheritance in most situations. I'll get to that shortly, but knowing how inheritance works is helpful.

Inherit from a Parent Class

You define only what you need to add or change in the new class, and this overrides the behavior of the old class. The original class is called a *parent*, *superclass*, or *base class*; the new class is called a *child*, *subclass*, or *derived class*. These terms are interchangeable in object-oriented programming.

So, let's inherit something. In the next example, we define an empty class called `Car`. Next, we define a subclass of `Car` called `Yugo`.[3] You define a subclass by using the same `class` keyword but with the parent class name inside the parentheses (`class Yugo(Car)` here):

```
>>> class Car():
...     pass
...
```

3 An inexpensive but not-so-good car from the '80s.

```
>>> class Yugo(Car):
...     pass
...
```

You can check whether a class is derived from another class by using `issubclass()`:

```
>>> issubclass(Yugo, Car)
True
```

Next, create an object from each class:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

A child class is a specialization of a parent class; in object-oriented lingo, `Yugo` *is-a* `Car`. The object named `give_me_a_yugo` is an instance of class `Yugo`, but the object also inherits whatever a `Car` can do. In this case, `Car` and `Yugo` are as useful as deckhands on a submarine, so let's try new class definitions that actually do something:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...
```

Finally, make one object from each class and call the `exclaim()` method:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

Without doing anything special, `Yugo` inherits the `exclaim()` method from `Car`. In fact, `Yugo` says that it *is* a `Car` (generically true, but potentially specialized), which might lead to an identity crisis. Let's see what we can do about that.

Inheritance is appealing but can be overused. Years of object-oriented programming experience have shown that too much use of inheritance can make programs hard to manage. Instead, emphasizing other techniques like aggregation and composition is often recommended. We get to these alternatives in "Aggregation and Composition" on page 208.

Override a Method

As you just saw, a new class initially inherits everything from its parent class. Moving forward, you'll see how to replace or override a parent method. `Yugo` should probably

be different from `Car` in some way; otherwise, what's the point of defining a new class? Let's change how the `exclaim()` method works for a `Yugo`:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! You go not very far with me.")
...
```

Now make two objects from these classes:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

What do they say?

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! You go not very far with me.
```

In these examples, we override the `exclaim()` method. We can override any methods, including `__init__()`. Here's another example that uses a `Person` class. Let's make subclasses that represent doctors (`MDPerson`) and lawyers (`JDPerson`):

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

In these cases, the initialization method `__init__()` takes the same arguments as the parent `Person` class but stores the value of `name` differently inside the object instance:

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

Add a Method

The child class can also *add* a method that is not present in its parent class. Going back to the `Car` and `Yugo` classes, we'll define the new method `need_a_push()` for class `Yugo` only:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

Next, make a `Car` and a `Yugo`:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

A `Yugo` object can react to a `need_a_push()` method call:

```
>>> give_me_a_yugo.need_a_push()
A little help here?
```

But a generic `Car` object cannot:

```
>>> give_me_a_car.need_a_push()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'need_a_push'
```

At this point, a `Yugo` can do something that a `Car` cannot, and the distinct personality of a `Yugo` can emerge.

Get Help from Your Parent with super()

We saw how the child class could add or override a method from the parent. What if the class wants to call that parent method? "I'm glad you asked," says `super()`. Here, we define a new class called `EmailPerson` that represents a `Person` with an email address. First, our familiar `Person` definition:

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
```

Notice that the `__init__()` call in the following subclass has an additional `email` parameter:

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

When you define an `__init__()` method for your class, you're replacing the `__init__()` method of its parent class, and the latter is not called automatically anymore. As a result, we need to call it explicitly. Here's what's happening:

- The `super()` function gets the definition of the parent class, `Person`.
- The `__init__()` method calls the `Person.__init__()` method. It takes care of passing the `self` argument to the superclass, so you just need to give it any optional arguments. In our case, the only other argument `Person()` accepts is `name`.
- The `self.email = email` line is the new code that makes this `EmailPerson` different from a `Person`.

Moving on, let's make one of these creatures:

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

We should be able to access both the `name` and `email` attributes:

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

Why didn't we just define our new class as follows?

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         self.name = name
...         self.email = email
```

We could have done that, but it would have defeated our use of inheritance. We use `super()` to make `Person` do its work, the same as a plain `Person` object would. There's another benefit: if the definition of `Person` changes in the future, using `super()` will ensure that the attributes and methods that `EmailPerson` inherits from `Person` will reflect the change.

Use `super()` when the child is doing something its own way but still needs something from the parent (as in real life).

Use Multiple Inheritance

You've just seen some class examples with no parent class, and some with one. Actually, objects can inherit from multiple parent classes.

If your class refers to a method or attribute that it doesn't have, Python will look in all the parents. What if more than one of them has something with that name? Who wins?

Unlike inheritance in people, where a dominant gene wins no matter who it came from, inheritance in Python depends on *method resolution order*. Each Python class has a special method called `mro()` that returns a list of the classes that would be visited to find a method or attribute for an object of that class. A similar attribute, called `__mro__`, is a tuple of those classes. Like a sudden-death playoff, the first one wins.

Here, we define a top `Animal` class, two child classes (`Horse` and `Donkey`), and then two derived from these:[4]

```
>>> class Animal:
...     def says(self):
            return 'I speak!'
...
>>> class Horse(Animal):
...     def says(self):
...         return 'Neigh!'
...
>>> class Donkey(Animal):
...     def says(self):
...         return 'Hee-haw!'
...
>>> class Mule(Donkey, Horse):
...     pass
...
>>> class Hinny(Horse, Donkey):
...     pass
...
```

If we look for a method or attribute of a `Mule`, Python will look at the following sources, in this order:

1. The object itself (of type `Mule`)
2. The object's class (`Mule`)
3. The class's first parent class (`Donkey`)
4. The class's second parent class (`Horse`)
5. The grandparent class (`Animal`)

4 A mule has a father donkey and mother horse; a hinny has a father horse and mother donkey.

The process is much the same for a `Hinny`, but with `Horse` before `Donkey`:

```
>>> Mule.mro()
[<class '__main__.Mule'>, <class '__main__.Donkey'>,
<class '__main__.Horse'>, <class '__main__.Animal'>,
<class 'object'>]
>>> Hinny.mro()
[<class '__main__.Hinny'>, <class '__main__.Horse'>,
<class '__main__.Donkey'>, <class '__main__.Animal'>,
class 'object'>]
```

So what do these fine beasts say?

```
>>> mule = Mule()
>>> hinny = Hinny()
>>> mule.says()
'hee-haw'
>>> hinny.says()
'neigh'
```

We list the parent classes in (father, mother) order, so they talk like their dads.

If the `Horse` and `Donkey` did not have a `says()` method, the mule or hinny would have used the grandparent `Animal` class's `says()` method, and returned `I speak!`.

Include Mixins

You may include an extra parent class in your class definition, but as a helper only. If you control all the class code, it doesn't share any methods with the other parent classes, and avoids the method resolution ambiguity that I mentioned in the previous section.

Such a parent class is sometimes called a *mixin* class. Uses might include "side" tasks like logging. Here's a mixin that pretty-prints an object's attributes:

```
>>> class PrettyMixin():
...     def dump(self):
...         import pprint
...         pprint.pprint(vars(self))
...
>>> class Thing(PrettyMixin):
...     pass
...
>>> t = Thing()
>>> t.name = "Nyarlathotep"
>>> t.feature = "ichor"
>>> t.age = "eldritch"
>>> t.dump()
{'age': 'eldritch', 'feature': 'ichor', 'name': 'Nyarlathotep'}
```

In self Defense

One criticism of Python (besides the use of whitespace) is the need to include `self` as the first argument to instance methods (the kind of method you've seen in the previous examples). Python uses the `self` argument to find the right object's attributes and methods. A defense against this criticism is that Python always favors the explicit, visible approach over the implicit, esoteric one. For an example, I'll show how you would call an object's method, and what Python does behind the scenes.

Remember the `Car` class from earlier examples? Let's call its `exclaim()` method again:

```
>>> a_car = Car()
>>> a_car.exclaim()
I'm a Car!
```

Here's what Python does, under the hood:

- Look up the class (`Car`) of the object `a_car`
- Pass the object `a_car` to the `exclaim()` method of the `Car` class as the `self` parameter

Just for fun, you can even run the code this way yourself, and it will work the same as the normal (`a_car.exclaim()`) syntax:

```
>>> Car.exclaim(a_car)
I'm a Car!
```

However, there's never a reason to use that lengthier style. I mean, come on, Guido did all that work.

Attribute Access

In Python, object attributes and methods are normally public (visible outside the object), and you're expected to behave yourself (this is sometimes called a *consenting adults* policy). Let's compare the direct approach with some alternatives.

Direct Access

As you've seen, you can get and set attribute values directly:

```
>>> class Duck:
...     def __init__(self, input_name):
...         self.name = input_name
...
>>> fowl = Duck('Daffy')
>>> fowl.name
'Daffy'
```

But if the attribute is visible and unprotected, anyone can change it:

```
>>> fowl.name = 'Daphne'
>>> fowl.name
'Daphne'
```

The next two sections show ways to get some privacy for attributes that you don't want anyone to stomp by accident.

Getters and Setters

Some object-oriented languages support private object attributes that can't be accessed directly from the outside. Programmers then may need to write *getter* and *setter* methods to read and write the values of such private attributes.

Python doesn't have private attributes, but you can write getters and setters with obfuscated attribute names to get a little privacy. (The best solution is to use *properties*, described in the next section.)

In the following example, we define a `Duck` class with a single instance attribute called `_hidden_name`. (I used an initial `_` to hint that this attribute should be somewhat private.) We don't want people to access this directly, so we define two methods: a getter (`get_name()`) and a setter (`set_name()`). I've added a `print()` statement to each method to show when it's being called:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self._hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self._hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self._hidden_name = input_name

>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
>>> don.set_name('Donna')
inside the setter
>>> don.get_name()
inside the getter
'Donna'
```

Properties for Attribute Access

The Pythonic solution for attribute privacy is to use *properties*.

You can do this in two ways. The first way is to add *`name = property(get_name, set_name)`* as the final line of our previous `Duck` class definition:

```
>>> class Duck():
>>>     def __init__(self, input_name):
>>>         self._hidden_name = input_name
>>>     def get_name(self):
>>>         print('inside the getter')
>>>         return self._hidden_name
>>>     def set_name(self, input_name):
>>>         print('inside the setter')
>>>         self._hidden_name = input_name
>>>     name = property(get_name, set_name)
```

The old getter and setter still work:

```
>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
>>> don.set_name('Donna')
inside the setter
>>> don.get_name()
inside the getter
'Donna'
```

But now you can also use the property `name` to get and set the hidden name:

```
>>> don = Duck('Donald')
>>> don.name
inside the getter
'Donald'
>>> don.name = 'Donna'
inside the setter
>>> don.name
inside the getter
'Donna'
```

In the second approach, you add decorators and replace the getter and setter method names (`get_name` and `set_name`) with `name`:

- `@property`, which goes before the *getter* method
- *`@name`*`.setter`, which goes before the *setter* method

Here's how they look in the code:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self._hidden_name = input_name
```

```
...     @property
...     def name(self):
...         print('inside the getter')
...         return self._hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self._hidden_name = input_name
```

You can still access `name` as though it were an attribute:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

If anyone guessed that we called our attribute `_hidden_name`, they could still read and write it directly as `fowl._hidden_name`. In "Name Mangling for Privacy" on page 199, you'll see how Python provides a special way to hide attribute names.

Properties for Computed Values

In the previous examples, we used the `name` property to refer to a single attribute (`hidden_name`) stored within the object.

A property can also return a *computed value*. Let's define a `Circle` class that has a `radius` attribute and a computed `diameter` property:

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
...     @property
...     def diameter(self):
...         return 2 * self.radius
...
```

Create a `Circle` object with an initial value for its `radius`:

```
>>> c = Circle(5)
>>> c.radius
5
```

We can refer to `diameter` as if it were an attribute such as `radius`:

```
>>> c.diameter
10
```

Here's the fun part: we can change the `radius` attribute at any time, and the diameter property will be computed from the current value of `radius`:

```
>>> c.radius = 7
>>> c.diameter
14
```

If you don't specify a setter property for an attribute, you can't set it from the outside. This is handy for read-only attributes:

```
>>> c.diameter = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Using a property over direct attribute access has one more advantage: if you ever change the definition of the attribute, you need to fix only the code within the class definition, not in all the callers.

Name Mangling for Privacy

In the `Duck` class example a little earlier, we called our (not completely) hidden attribute `_hidden_name`. Python has a naming convention for attributes that should not be visible outside their class definition: begin with two underscores (`__`).

Let's rename `hidden_name` to `__name`, as demonstrated here:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.__name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.__name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.__name = input_name
...
```

Take a moment to see whether the code still works:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Looks good. And you can't access the `__name` attribute:

```
>>> fowl.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

This naming convention doesn't make the attribute completely private, but Python does *mangle* its name to make it unlikely for external code to stumble upon it. If you're curious and promise not to tell everyone,[5] here's what the name becomes:

```
>>> fowl._Duck__name
'Donald'
```

Notice that the code doesn't print `inside the getter`. Although this isn't perfect protection, name mangling discourages accidental or intentional direct access to the attribute.

Class and Object Attributes

You can assign attributes to classes, and they'll be inherited by their child objects:

```
>>> class Fruit:
...     color = 'red'
...
>>> blueberry = Fruit()
>>> Fruit.color
'red'
>>> blueberry.color
'red'
```

But changing the value of the attribute in the child object doesn't affect the class attribute:

```
>>> blueberry.color = 'blue'
>>> blueberry.color
'blue'
>>> Fruit.color
'red'
```

If you change the class attribute later, it won't affect existing child objects:

```
>>> Fruit.color = 'orange'
>>> Fruit.color
'orange'
>>> blueberry.color
'blue'
```

5 Can you keep a secret? Obviously, I can't.

But the change will affect new ones:

```
>>> new_fruit = Fruit()
>>> new_fruit.color
'orange'
```

Method Types

Some methods are part of the class itself, some are part of the objects that are created from that class, and some are none of the above:

- If there's no preceding decorator, it's an *instance method*, and its first argument should be `self` to refer to the individual object itself.
- If there's a preceding `@classmethod` decorator, it's a *class method*, and its first argument should be `cls` (or anything, just not the reserved word `class`), referring to the class itself.
- If there's a preceding `@staticmethod` decorator, it's a *static method*, and its first argument isn't an object or class.

The following sections have some details.

Instance Methods

When you see an initial `self` argument in methods within a class definition, it's an *instance method*. These are the types of methods that you would normally write when creating your own classes. The first parameter of an instance method is `self`, and Python passes the object to the method when you call it. These are the ones that you've seen so far.

Class Methods

In contrast, a *class method* affects the class as a whole. Any change you make to the class affects all its objects. Within a class definition, a preceding `@classmethod` decorator indicates that the following function is a class method. Also, the first parameter to the method is the class itself. The Python tradition is to call the parameter `cls`, because `class` is a reserved word and can't be used here. Let's define a class method for `A` that counts the number of object instances that have been made from it:

```
>>> class A():
...     count = 0
...     def __init__(self):
...         A.count += 1
...     def exclaim(self):
...         print("I'm an A!")
...     @classmethod
...     def kids(cls):
...         print("A has", cls.count, "little objects.")
```

```
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
A has 3 little objects.
```

Notice that we refer to `A.count` (the class attribute) in `__init__()` rather than `self.count` (which would be an object instance attribute). In the `kids()` method, we use `cls.count`, but we could just as well have used `A.count`.

Static Methods

A third type of method in a class definition affects neither the class nor its objects; it's just there for convenience instead of floating around on its own. It's a *static method*, preceded by a `@staticmethod` decorator, with no initial `self` or `cls` parameter. Here's an example that serves as a commercial for the class `CoyoteWeapon`:

```
>>> class CoyoteWeapon():
...     @staticmethod
...     def commercial():
...         print('This CoyoteWeapon has been brought to you by Acme')
...
>>>
>>> CoyoteWeapon.commercial()
This CoyoteWeapon has been brought to you by Acme
```

Notice that we don't need to create an object from class `CoyoteWeapon` to access this method. Very class-y.

Duck Typing

Python has a loose implementation of *polymorphism*; it applies the same operation to different objects, based on the method's name and arguments, regardless of their class.

Let's use the same `__init__()` initializer for all three `Quote` classes now, but add two new functions:

- `who()` returns the value of the saved `person` string.
- `says()` returns the saved `words` string with the specific punctuation.

And here they are in action:

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
```

```
...         return self.person
...     def says(self):
...         return self.words + '.'
...
>>> class QuestionQuote(Quote):
...     def says(self):
...         return self.words + '?'
...
>>> class ExclamationQuote(Quote):
...     def says(self):
...         return self.words + '!'
...
>>>
```

We don't change how `QuestionQuote` or `ExclamationQuote` are initialized, so we don't override their `__init__()` methods. Python then automatically calls the `__init__()` method of the parent class `Quote` to store the instance variables `person` and `words`. That's why we can access `self.words` in objects created from the subclasses `QuestionQuote` and `ExclamationQuote`.

Next up, let's make some objects:

```
>>> hunter = Quote('Elmer Fudd', "I'm hunting wabbits")
>>> print(hunter.who(), 'says:', hunter.says())
Elmer Fudd says: I'm hunting wabbits.

>>> hunted1 = QuestionQuote('Bugs Bunny', "What's up, doc")
>>> print(hunted1.who(), 'says:', hunted1.says())
Bugs Bunny says: What's up, doc?

>>> hunted2 = ExclamationQuote('Daffy Duck', "It's rabbit season")
>>> print(hunted2.who(), 'says:', hunted2.says())
Daffy Duck says: It's rabbit season!
```

Three versions of the `says()` method provide different behavior for the three classes. This is traditional polymorphism in object-oriented languages. Python goes a little further and lets you run the `who()` and `says()` methods of *any* objects that have them. Let's define a class called `BabblingBrook` that has no relation to our previous woodsy hunter and huntees (descendants of the `Quote` class):

```
>>> class BabblingBrook():
...     def who(self):
...         return 'Brook'
...     def says(self):
...         return 'Babble'
...
>>> brook = BabblingBrook()
```

Now run the `who()` and `says()` methods of various objects, one (`brook`) completely unrelated to the others:

```
>>> def who_says(obj):
...     print(obj.who(), 'says', obj.says())
...
>>> who_says(hunter)
Elmer Fudd says I'm hunting wabbits.
>>> who_says(hunted1)
Bugs Bunny says What's up, doc?
>>> who_says(hunted2)
Daffy Duck says It's rabbit season!
>>> who_says(brook)
Brook says Babble
```

This behavior is sometimes called *duck typing* (Figure 11-1). The term comes from this old saying:

> If it walks like a duck and quacks like a duck, it's a duck.
>
> —A Wise Person

Who are we to argue with a wise saying about ducks?

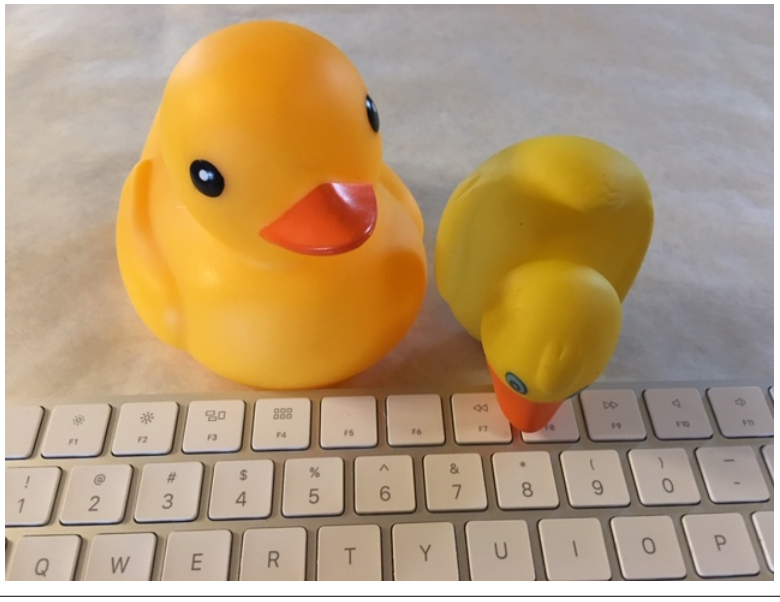

Figure 11-1. Duck typing is not hunt-and-peck

Magic Methods

You can now create and use basic objects. What you'll learn in this section might surprise you—in a good way.

When you type something such as `a = 3 + 8`, how do the integer objects with values `3` and `8` know how to implement `+`? Or, if you type `name = "Daffy" + " " + "Duck"`, how does Python know that `+` now means to concatenate these strings? And how do `a` and `name` know how to use `=` to get the result? You can access these operators by using Python's *special methods* (or, more dramatically, *magic methods*).

The names of these methods begin and end with double underscores (`__`). Why? They're unlikely to have been chosen by programmers as variable names. You've already seen one: `__init__()` initializes a newly created object from its class definition and any arguments that were passed in. You've also seen how dunder naming helps mangle class attribute names as well as methods.

Suppose that you have a simple `Word` class, and you want an `equals()` method that compares two words but ignores case. That is, a `Word` containing the value `ha` would be considered equal to one containing `HA`.

The example that follows is a first attempt, with a normal method we're calling `equals()`. The attribute `self.text` is the text string that this `Word` object contains, and the `equals()` method compares `self.text` with the text string of `word2` (another `Word` object):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...
...     def equals(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Then make three `Word` objects from three text strings:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
```

When strings `ha` and `HA` are compared to lowercase, they should be equal:

```
>>> first.equals(second)
True
```

But the string `eh` will not match `ha`:

```
>>> first.equals(third)
False
```

We define the method `equals()` to do this lowercase conversion and comparison. It would be nice to just say `if first == second`, just like Python's built-in types. So, let's do that. We change the `equals()` method to the special name `__eq__()` (you'll see why in a moment):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Let's see whether this change works:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
>>> first == second
True
>>> first == third
False
```

Magic! All we needed was the Python's special method name for testing equality, `__eq__()`. Tables 11-1 and 11-2 list the names of the most useful magic methods.

Table 11-1. Magic methods for comparison

Method	Description
`__eq__( self, other )`	*self == other*
`__ne__( self, other )`	*self != other*
`__lt__( self, other )`	*self < other*
`__gt__( self, other )`	*self > other*
`__le__( self, other )`	*self <= other*
`__ge__( self, other )`	*self >= other*

Table 11-2. Magic methods for math

Method	Description
`__add__( self, other )`	*self + other*
`__sub__( self, other )`	*self – other*
`__mul__( self, other )`	*self * other*
`__floordiv__( self, other )`	*self // other*
`__truediv__( self, other )`	*self / other*
`__mod__( self, other )`	*self % other*
`__pow__( self, other )`	*self ** other*

You aren't restricted to use the math operators such as + (magic method `__add__()`) and - (magic method `__sub__()`) with numbers. For instance, Python string objects use + for concatenation and * for duplication. There are many more, documented online at "Special method names". The most common among them are presented in Table 11-3.

Table 11-3. Other, miscellaneous magic methods

Method	Description
`__str__(`*self*`)`	`str(`*self*`)`
`__repr__(`*self*`)`	`repr(`*self*`)`
`__len__(`*self*`)`	`len(`*self*`)`

Besides `__init__()`, you might find yourself using `__str__()` the most in your own methods. It's how you print your object. It's used by `print()`, `str()`, and the string formatters, which you can read about in Chapter 17. The interactive interpreter uses the `__repr__()` function to echo variables to output. If you fail to define either `__str__()` or `__repr__()`, you get Python's default string version of your object:

```
>>> first = Word('ha')
>>> first
<__main__.Word object at 0x1006ba3d0>
>>> print(first)
<__main__.Word object at 0x1006ba3d0>
```

Let's add the `__str__()` and `__repr__()` methods to the `Word` class to make it prettier:

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...     def __str__(self):
...         return self.text
...     def __repr__(self):
...         return 'Word("'  + self.text  + '")'
...
>>> first = Word('ha')
>>> first          # uses __repr__
Word("ha")
>>> print(first)   # uses __str__
ha
```

When you're creating your own classes, it can be tedious to write all these dunder methods. See "Dataclasses" on page 211 for a time saver. To explore even more special methods, check out the Python documentation.

Aggregation and Composition

Inheritance is a good technique to use when you want a child class to act like its parent class most of the time (when child *is-a* parent). Building elaborate inheritance hierarchies is tempting, but this couples child classes tightly to details of the parent class.

Sometimes *composition* or *aggregation* make more sense. What's the difference? In composition, one thing is part of another. A duck *is-a* bird (inheritance), but *has-a* tail (composition). A tail is not a kind of duck, but part of a duck. In this next example, let's make `bill` and `tail` objects and provide them to a new `duck` object:

```
>>> class Bill():
...     def __init__(self, description):
...         self.description = description
...
>>> class Tail():
...     def __init__(self, length):
...         self.length = length
...
>>> class Duck():
...     def __init__(self, bill, tail):
...         self.bill = bill
...         self.tail = tail
...     def about(self):
...         print('This duck has a', self.bill.description,
...             'bill and a', self.tail.length, 'tail')
...
>>> a_tail = Tail('long')
>>> a_bill = Bill('wide orange')
>>> duck = Duck(a_bill, a_tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

Aggregation expresses relationships but is a little looser: one thing *uses* another, but both exist independently. A duck *uses* a lake, but one is not a part of the other.

When to Use Objects or Something Else

Here are some guidelines for deciding whether to put your code and data in a class, module (discussion coming in Chapter 12), or something entirely different:

- Objects are most useful when you need individual instances that have similar behavior (methods) but differ in their internal states (attributes).
- Classes support inheritance; modules don't.

- If you want only one of something, a module might be best. No matter how many times a Python module is referenced in a program, only one copy is loaded. (Java and C++ programmers: you can use a Python module as a *singleton*.)
- Alternatively, a class-based way to enforce a single behavior is to use the *Borg pattern* (named after aliens in *Star Trek: The Next Generation*).
- If you have variables that contain multiple values and can be passed as arguments to multiple functions, defining them as classes might be better. For example, you might use a dictionary with keys such as `size` and `color` to represent a color image. You could create a different dictionary for each image in your program, and pass them as arguments to functions such as `scale()` or `transform()`. This can get messy as you add keys and functions. It's more coherent to define an `Image` class with attributes `size` or `color` and methods `scale()` and `transform()`. Then all the data and methods for a color image are defined in one place.
- Use the simplest solution to the problem. A dictionary, list, or tuple is simpler, smaller, and faster than a module, which is usually simpler than a class.

  Here's some advice:

  > Avoid overengineering datastructures. Tuples are better than objects (try namedtuple, too, though). Prefer simple fields over getter/setter functions. Built-in datatypes are your friends. Use more numbers, strings, tuples, lists, sets, dicts. Also check out the collections library, especially `deque`.
  >
  > —Guido van Rossum

- As I mentioned a bit earlier, a newer alternative is the *dataclass*, described in "Dataclasses" on page 211.

Named Tuples

Because Guido just mentioned them and I haven't yet, this is a good place to talk about named tuples. A *named tuple* is a subclass of tuples with which you can access values by name (with `.name`) as well as by position (with `[ offset ]`).

Let's take the example from the previous section and convert the `Duck` class to a named tuple, with `bill` and `tail` as simple string attributes. We'll call the `namedtuple()` function with two arguments:

- The name
- A string of the field names, separated by spaces

Named tuples are not automatically supplied with Python, so you need to load a module before using them. We do that in the first line of the following example:

```
>>> from collections import namedtuple
>>> Duck = namedtuple('Duck', 'bill tail')
>>> duck = Duck('wide orange', 'long')
>>> duck
Duck(bill='wide orange', tail='long')
>>> duck.bill
'wide orange'
>>> duck.tail
'long'
```

You can also make a named tuple from a dictionary:

```
>>> parts = {'bill': 'wide orange', 'tail': 'long'}
>>> duck2 = Duck(**parts)
>>> duck2
Duck(bill='wide orange', tail='long')
```

In the preceding code, take a look at `**parts`. This is a *keyword argument*. It extracts the keys and values from the `parts` dictionary and supplies them as arguments to `Duck()`. It has the same effect as this:

```
>>> duck2 = Duck(bill = 'wide orange', tail = 'long')
```

Named tuples are immutable, but you can replace one or more fields and return another named tuple:

```
>>> duck3 = duck2._replace(tail='magnificent', bill='crushing')
>>> duck3
Duck(bill='crushing', tail='magnificent')
```

We could have defined duck as a dictionary:

```
>>> duck_dict = {'bill': 'wide orange', 'tail': 'long'}
>>> duck_dict
{'tail': 'long', 'bill': 'wide orange'}
```

You can add fields to a dictionary:

```
>>> duck_dict['color'] = 'green'
>>> duck_dict
{'color': 'green', 'tail': 'long', 'bill': 'wide orange'}
```

But not to a named tuple:

```
>>> duck.color = 'green'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute 'color'
```

To recap, here are some of the pros of a named tuple:

- It looks and acts like an immutable object.
- It is more space and time efficient than objects.
- You can access attributes by using dot notation instead of dictionary-style square brackets.
- You can use a named tuple as a dictionary key.

Dataclasses

Many people like to create objects mainly to store data (as object attributes), not so much behavior (methods). You just saw how named tuples can be an alternative data store. Python 3.7 introduced dataclasses.

Here's a plain old object with a single `name` attribute:

```
>> class TeenyClass():
...     def __init__(self, name):
...         self.name = name
...
>>> teeny = TeenyClass('itsy')
>>> teeny.name
'itsy'
```

Doing the same with a dataclass looks a little different:

```
>>> from dataclasses import dataclass
>>> @dataclass
... class TeenyDataClass:
...     name: str
...
>>> teeny = TeenyDataClass('bitsy')
>>> teeny.name
'bitsy'
```

Besides needing a `@dataclass` decorator, you define the class's attributes with *variable annotations* of the form *`name: type`* or *`name: type = val`*, like `color: str` or `color: str = "red"`. (Also see Chapter 14 on type hints.) The *`type`* can be any Python object type, including classes you've created, not just the built-in ones like `str` or `int`.

When you're creating the dataclass object, you provide the arguments in the order in which they were specified in the class, or use named arguments in any order:

```
>>> from dataclasses import dataclass
>>> @dataclass
... class AnimalClass:
...     name: str
...     habitat: str
```

```
...     teeth: int = 0
...
>>> snowman = AnimalClass('yeti', 'Himalayas', 46)
>>> duck = AnimalClass(habitat='lake', name='duck')
>>> snowman
AnimalClass(name='yeti', habitat='Himalayas', teeth=46)
>>> duck
AnimalClass(name='duck', habitat='lake', teeth=0)
```

`AnimalClass` defines a default value for its `teeth` attribute, so we don't need to provide it when making a `duck`.

You can refer to the object attributes like any other object's:

```
>>> duck.habitat
'lake'
>>> snowman.teeth
46
```

I haven't mentioned that dataclasses define many features for you automatically, so you don't have to write them, and this can be a big time saver. See "Data Classes in Python 3.7+" by Geir Arne Hjelle or the official (heavy) docs for many examples.

Attrs

You've seen how to create classes and add attributes, and how they can involve a lot of typing—steps like defining `__init__()`, assigning its arguments to `self` counterparts, and creating all those dunder methods like `__str__()`. Named tuples and dataclasses are alternatives in the standard library that may be easier when you mainly want to create a data collection.

"The One Python Library Everyone Needs" by Glyph Lefkowitz compares plain classes, named tuples, and dataclasses. It recommends the third-party package attrs for many reasons—less typing, data validation, and more. Take a look and see whether you prefer it to the built-in solutions.

Review/Preview

Objects! Whew. That's a lot to absorb. But you should know that you may never need to define a class! It may sound heretical,[6] but you can survive in Python by using all the other built-in data and code structures.

In fact, some of the reasons you would define an object class can also be handled by modules and packages, which are, conveniently, in the next chapter.

6 But see my earlier quote from Guido himself.

Practice

11.1 Make a class called `Thing` with no contents and print it. Then create an object called `example` from this class and also print it. Are the printed values the same or different?

11.2 Make a new class called `Thing2` and assign the value `abc` to a class attribute called `letters`. Print `letters`.

11.3 Make yet another class called, of course, `Thing3`. This time, assign the value `xyz` to an instance (object) attribute called `letters`. Print `letters`.

11.4 Make a class called `Element`, with instance attributes `name`, `symbol`, and `number`. Create an object of this class with the values `Hydrogen`, `H`, and `1`.

11.5 Make a dictionary with these keys and values: `'name': 'Hydrogen', 'symbol': 'H', 'number': 1`. Then create an object called `hydrogen` from class `Element` by using this dictionary.

11.6 For the `Element` class, define a method called `dump()` that prints the values of the object's attributes (`name`, `symbol`, and `number`). Create the `hydrogen` object from this new definition and use `dump()` to print its attributes.

11.7 Call `print(hydrogen)`. In the definition of `Element`, change the name of the method `dump()` to `__str__`, create a new `hydrogen` object, and call `print(hydrogen)` again.

11.8 Modify `Element` to make the attributes `name`, `symbol`, and `number` private. Define a getter property for each to return its value.

11.9 Define three classes: `Bear`, `Rabbit`, and `Octothorpe`. For each, define only one method: `eats()`. This should return `berries` (`Bear`), `clover` (`Rabbit`), or `campers` (`Octothorpe`). Create one object from each and print what it eats.

11.10 Define these classes: `Laser`, `Claw`, and `SmartPhone`. Each has only one method: `does()`. This returns `disintegrate` (`Laser`), `crush` (`Claw`), or `ring` (`SmartPhone`). Then define the class `Robot` that has one instance (object) of each of these. Define a `does()` method for the `Robot` that prints what its component objects do.

CHAPTER 12
Modules and Packages

Information about the package is as important as the package itself.
—Frederick W. Smith

During your bottom-up climb, you've progressed from built-in data types to constructing ever-larger data and code structures. In this chapter, you finally learn to write realistic whole programs in Python. You'll write your own modules and learn how to use others from Python's standard library and other sources.

This book is organized in a hierarchy: words, sentences, paragraphs, and chapters. Otherwise, it would become unreadable pretty quickly.[1] Code has a roughly similar bottom-up organization: data types are like words; expressions and statements are like sentences; functions are like paragraphs; and modules are like chapters. To continue the analogy, when I say that something is explained in Chapter 8 in this book, in programming that's like referring to code in another module.

Modules and the import Statement

We'll create and use Python code in more than one file. A *module* is just a file of any Python code. You don't need to do anything special; any Python code can be used as a module by others.

We refer to code of other modules by using the Python `import` statement. This makes the code and variables in the imported module available to your program.

1 At least, a little less readable than it already is.

Import a Module

Python lets you import code from separate modules in different ways:

- The simplest use of the `import` statement is `import` *`module`*, where *`module`* is the name of another Python file, without the *.py* extension.
- To access only the function `do_something()` in a module called `wonders`, use `from wonders import do_something`.
- To access everything in `wonders`, use `from wonders import *`. This is usually discouraged, because you'll get, well, everything, as variables now strewn through your calling program, possibly including stuff that you don't know or want.

Let's say you and a few others want something fast for lunch but don't want a long discussion, and you always end up picking what the loudest person wants anyhow. Let the computer decide! Let's write a module with a single function that returns a random fast-food choice, and a main program that calls it and prints the choice. The Python standard library contains a module called *random.py*, and within it is a function called `choice()`:

A module that imports it is shown in Example 12-1.

Example 12-1. fast.py

```
from random import choice

places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
     "Wendys", "Arbys", "Pizza Hut"]

def pick():  # see the docstring below?
    """Return random fast food place"""
    return choice(places)
```

And Example 12-2 shows the main program that imports the module.

Example 12-2. lunch.py

```
import fast

place = fast.pick()
print("Let's go to", place)
```

If you have these two files in the same directory and instruct Python to run *lunch.py* as the main program, Python will access the `fast` module and run its `pick()` function. We wrote this version of `pick()` to return a random result from a list of strings, so that's what the main program will get back and print:

```
$ python lunch.py
Let's go to Burger King
$ python lunch.py
Let's go to Pizza Hut
$ python lunch.py
Let's go to Arbys
```

We're using imports in two places:

- The main program *lunch.py* imports our new module `fast`.
- The module file *fast.py* imports the `choice` function from Python's standard library module named `random`.

We also use imports in two ways in our main program and our module:

- In the first case, we import the entire `fast` module but need to use `fast` as a prefix to `pick()`. After this `import` statement, everything in *fast.py* is available to the main program, as long as we prepend `fast.` to its name. By *qualifying* the contents of a module with the module's name, we avoid any nasty naming conflicts. There could be a `pick()` function in another module, and we would not call it by mistake.
- In the second case, we're within a module and know that nothing else named `choice` is here, so we import the `choice()` function from the `random` module directly.

We could have written *fast.py*, as shown in Example 12-3, importing `random` within the `pick()` function instead of at the top of the file.

Example 12-3. fast2.py

```
places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
     "Wendys", "Arbys", "Pizza Hut"]

def pick():
    import random
    return random.choice(places)
```

As with many aspects of programming, use the style that seems clearest to you. The module-qualified name (`random.choice`) is safer but requires a little more typing.

Consider importing from outside the function if the imported code might be used in more than one place, and from inside if you know its use will be limited. Some people prefer to put all their imports at the top of the file, just to make all the dependencies of their code explicit. Either way works.

Import a Module with Another Name

In our main *lunch.py* program, we call `import fast`. But what if you:

- Have another module named `fast` somewhere?
- Want to use a name that is more mnemonic?
- Caught your fingers in a door and want to minimize typing?

In these cases, you can import using an *alias*, as shown in Example 12-4. Let's use the module alias `f`.

Example 12-4. fast3.py

```
import fast as f
place = f.pick()
print("Let's go to", place)
```

Import Only What You Want from a Module

You can import a whole module or just parts of it. You just saw the latter: we wanted only the `choice()` function from the `random` module.

As with the module itself, you can use an alias for each variable that you import.

Let's redo our example a few more times. First, import `pick()` from the `fast` module with its original name (Example 12-5).

Example 12-5. fast4.py

```
from fast import pick
place = pick()
print("Let's go to", place)
```

Now import it as `who_cares` (Example 12-6).

Example 12-6. fast5.py

```
from fast import pick as who_cares
place = who_cares()
print("Let's go to", place)
```

Packages

We went from single lines of code, to multiline functions, to standalone programs, to multiple modules in the same directory. If you don't have many modules, the same directory works fine.

To allow Python applications to scale even more, you can organize modules into file and module hierarchies called packages. A *package* is just a subdirectory that contains *.py* files. And you can go more than one level deep, with directories inside those.

We just wrote a module that chooses a fast-food place. Let's add a similar module to dispense life advice. We'll make one new main program called *questions.py* in our current directory. Now make a subdirectory named *choices* and put two modules in it: *fast.py* and *advice.py*. Each module has a function that returns a string.

The main program (*questions.py*) has an extra import and line (Example 12-7).

Example 12-7. questions.py

```
from choices import fast, advice

print("Let's go to", fast.pick())
print("Should we take out?", advice.give())
```

That `from choices import fast, advice` makes Python look for a package (directory) named *choices*, starting under your current directory. Inside *choices* Python looks for the files *fast.py* and *advice.py*.

The first module (*choices/fast.py*) is the same code as before, just moved into the *choices* directory (Example 12-8).

Example 12-8. choices/fast.py

```
from random import choice

places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
     "Wendys", "Arbys", "Pizza Hut"]

def pick():
    """Return random fast food place"""
    return choice(places)
```

The second module (*choices/advice.py*) is new, but it works a lot like its fast-food relative (Example 12-9).

Example 12-9. choices/advice.py

```
from random import choice

answers = ["Yes!", "No!", "Reply hazy", "Sorry, what?"]

def give():
    """Return random advice"""
    return choice(answers)
```

Run the main *questions.py* program (from your current directory, not in *sources*) to see what happens:

```
$ python questions.py
Let's go to KFC
Should we take out? Yes!
$ python questions.py
Let's go to Wendys
Should we take out? Reply hazy
$ python questions.py
Let's go to McDonalds
Should we take out? Reply hazy
```

The Module Search Path

I just said that Python looks under your current directory for the subdirectory *choices* and its modules. Python looks in other places as well, and you can control this.

Earlier, we imported the `choice()` function from the standard library's `random` module. That isn't in your current directory, so Python needs to look elsewhere also.

To see all the places that your Python interpreter looks, import the standard `sys` module and use its `path` list. This is a list of directory names and ZIP archive files that Python searches in order to find modules to import.

You can access and modify this list. Here's the value of `sys.path` for Python 3.13 on my Mac:

```
>>> import sys
>>> for place in sys.path:
...     print(place)
...

/Library/Frameworks/Python.framework/Versions/3.13/lib/python313.zip
/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13
/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages
```

That initial blank output line is the empty string `''`, which stands for the current directory. If `''` is first in `sys.path`, Python looks in the current directory first when you try to import something: `import fast` looks for *fast.py*. This is Python's usual setup. Also, when we made that subdirectory called *sources* and put Python files in it, they could be imported with `import sources` or `from sources import fast`.

The first match will be used. This means that if you define a module named `random` and it's in the search path before the standard library, you won't be able to access the standard library's `random` now.

You can modify the search path within your code. Let's say you want Python to look in the */my/modules* directory before any other:

```
>>> import sys
>>> sys.path.insert(0, "/my/modules")
```

Relative and Absolute Imports

In our examples so far, we imported our own modules from the following:

- The current directory
- The subdirectory *choices*
- The Python standard library

This works well until you have a local module with the same name as a standard one. Which do you want?

Python supports *absolute* or *relative* imports. The examples that you've seen so far are absolute imports. If you typed **`import rougarou`** for each directory in the search path, Python would look for a file named *rougarou.py* (a module) or a directory named *rougarou* (a package).

- If *rougarou.py* is in the same directory as your calling problem, you can import it *relative* to your location with `from . import rougarou`.
- If the file is in the directory above you, use `from .. import rougarou`.
- If the file is under a sibling directory called *creatures*, use `from ..creatures import rougarou`.

The `.` and `..` notations were borrowed from Unix's shorthand for *current directory* and *parent directory*.

For a good discussion of Python import problems that you may run into, see "Traps for the Unwary in Python's Import System" by Alyssa Coghlan.

Namespace Packages

You've seen that you can package Python modules as follows:

- A single *module* (*.py* file)
- A *package* (directory containing modules, and possibly other packages)

You can also split a package across directories with *namespace packages*. Say you want a package called critters that will contain a Python module for each dangerous creature (real or imagined, supposedly with background info and protective hints). This might get large over time, and you'd like to subdivide these by geographic location. One option is to add location subpackages under critters and move the existing *.py* module files under them, but this would break code in other modules that import them. Instead, we can go *up* and do the following:

- Make new location directories above *critters*.
- Make cousin *critters* directories under these new parents.
- Move existing modules to their respective directories.

This needs some illustration. Say we start with this file layout:

```
critters
 ⌞ rougarou.py
 ⌞ wendigo.py
```

Normal imports of these modules would look like this:

```
from critters import wendigo, rougarou
```

Now if we decide on United States locations *north* and *south*, the files and directories will look like this:

```
north
 ⌞ critters
   ⌞ wendigo.py
south
 ⌞ critters
   ⌞ rougarou.py
```

If both *north* and *south* are in your module search path, you can import the modules as though they were still cohabiting a single-directory package:

```
from critters import wendigo, rougarou
```

Modules Versus Objects

When should you put your code into a module, and when into an object?

Modules and objects look similar in many ways. An object *or* module called `thing` with an internal data value called `stuff` would let you access the value as `thing.stuff`. The `stuff` value may have been defined when the module or class was created, or it may have been assigned later.

All the classes, functions, and global variables in a module are available to the outside. Objects can use properties and dunder (`__ …`) naming to hide or control access to their data attributes.

This means you can do this:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.pi = 3.0
>>> math.pi
3.0
```

Did you just ruin calculations for everyone on this computer? Yes! No, I'm kidding.[2] This did not affect the Python `math` module. You changed only the value of `pi` for the copy of the `math` module code imported by your calling program, and all evidence of your crimes will disappear when it finishes.

Only one copy of any module is imported by your program, even if you import it more than once. To use smart-sounding jargon, a module is a *singleton*. You can use it to save global code of interest to any code that imports it. This is similar to a class, which also has only one copy, although you can have many objects created from it. A class is used to define multiple object instances, each varying in some little way from its classmates.

Goodies in the Python Standard Library

One of Python's prominent claims is that it has "batteries included"—a large standard library of modules that perform many useful tasks. They are kept separate to avoid bloating the core language. When you're about to write some Python code, it's often worthwhile to first check whether a standard module already does what you want. It's surprising how often you encounter little gems in the standard library. Python also provides authoritative documentation for the modules, along with a tutorial. Doug Hellmann's website Python Module of the Week and book *The Python Standard Library by Example* (O'Reilly) are also useful guides.

Upcoming chapters in this book feature many of the standard modules that are specific to the web, systems, databases, and so on. In this section, I talk about some standard modules that have generic uses.

Handle Missing Keys with setdefault() and defaultdict()

You've seen that trying to access a dictionary with a nonexistent key raises an exception. Using the dictionary `get()` function to return a default value avoids an exception. The `setdefault()` function is like `get()` but also assigns an item to the dictionary if the key is missing:

2 Or am I? Bwa ha ha.

```
>>> periodic_table = {'Hydrogen': 1, 'Helium': 2}
>>> periodic_table
{'Hydrogen': 1, 'Helium': 2}
```

If the key is *not* already in the dictionary, the new value is used:

```
>>> carbon = periodic_table.setdefault('Carbon', 12)
>>> carbon
12
>>> periodic_table
{'Hydrogen': 1, 'Helium': 2, 'Carbon': 12}
```

If we try to assign a different default value to an *existing* key, the original value is returned and nothing is changed:

```
>>> helium = periodic_table.setdefault('Helium', 947)
>>> helium
2
>>> periodic_table
{'Hydrogen': 1, 'Helium': 2, 'Carbon': 12}
```

The `defaultdict()` function is similar but specifies the default value for any new key up front, when the dictionary is created. Its argument is a function. In this example, we pass the function `int`, which will be called as `int()` and return the integer `0`:

```
>>> from collections import defaultdict
>>> periodic_table = defaultdict(int)
```

Now any missing value will be an integer (`int`) with the value `0`:

```
>>> periodic_table['Hydrogen'] = 1
>>> periodic_table['Lead']
0
>>> periodic_table
defaultdict(<class 'int'>, {'Hydrogen': 1, 'Lead': 0})
```

The argument to `defaultdict()` is a function that returns the value to be assigned to a missing key. In the following example, `no_idea()` is executed to return a value when needed:

```
>>> from collections import defaultdict
>>>
>>> def no_idea():
...     return 'Huh?'
...
>>> bestiary = defaultdict(no_idea)
>>> bestiary['A'] = 'Abominable Snowman'
>>> bestiary['B'] = 'Basilisk'
>>> bestiary['A']
'Abominable Snowman'
>>> bestiary['B']
'Basilisk'
>>> bestiary['C']
'Huh?'
```

You can use the functions `int()`, `list()`, or `dict()` to return default empty values for those types: `int()` returns `0`, `list()` returns an empty list (`[]`), and `dict()` returns an empty dictionary (`{}`). If you omit the argument, the initial value of a new key will be set to `None`.

By the way, you can use `lambda` to define your default-making function right inside the call:

```
>>> bestiary = defaultdict(lambda: 'Huh?')
>>> bestiary['E']
'Huh?'
```

Using `int` is one way to make your own counter:

```
>>> from collections import defaultdict
>>> food_counter = defaultdict(int)
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     food_counter[food] += 1
...
>>> for food, count in food_counter.items():
...     print(food, count)
...
eggs 1
spam 3
```

In the preceding example, if `food_counter` had been a normal dictionary instead of a `defaultdict`, Python would have raised an exception every time we tried to increment the dictionary element `food_counter[food]` because it would not have been initialized. We would have needed to do some extra work, as shown here:

```
>>> dict_counter = {}
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     if not food in dict_counter:
...         dict_counter[food] = 0
...     dict_counter[food] += 1
...
>>> for food, count in dict_counter.items():
...     print(food, count)
...
spam 3
eggs 1
```

Count Items with Counter()

Speaking of counters, the standard library has one that does the work of the previous example and more:

```
>>> from collections import Counter
>>> breakfast = ['spam', 'spam', 'eggs', 'spam']
>>> breakfast_counter = Counter(breakfast)
```

```
>>> breakfast_counter
Counter({'spam': 3, 'eggs': 1})
```

The `most_common()` function returns all elements in descending order, or just the top `count` elements if given a count:

```
>>> breakfast_counter.most_common()
[('spam', 3), ('eggs', 1)]
>>> breakfast_counter.most_common(1)
[('spam', 3)]
```

You can combine counters. First, let's see again what's in `breakfast_counter`:

```
>>> breakfast_counter
>>> Counter({'spam': 3, 'eggs': 1})
```

This time, we make a new list called `lunch`, and a counter called `lunch_counter`:

```
>>> lunch = ['eggs', 'eggs', 'bacon']
>>> lunch_counter = Counter(lunch)
>>> lunch_counter
Counter({'eggs': 2, 'bacon': 1})
```

The first way we combine the two counters is by addition, using `+`:

```
>>> breakfast_counter + lunch_counter
Counter({'spam': 3, 'eggs': 3, 'bacon': 1})
```

As you might expect, you subtract one counter from another by using `-`. What's for breakfast but not for lunch?

```
>>> breakfast_counter - lunch_counter
Counter({'spam': 3})
```

OK, now what can we have for lunch that we can't have for breakfast?

```
>>> lunch_counter - breakfast_counter
Counter({'bacon': 1, 'eggs': 1})
```

Similar to sets in Chapter 9, you can get common items by using the intersection operator `&`:

```
>>> breakfast_counter & lunch_counter
Counter({'eggs': 1})
```

The intersection chooses the common element (`eggs`) with the lower count. This makes sense: breakfast offers only one egg, so that's the common count.

Finally, you can get all items by using the union operator `|`:

```
>>> breakfast_counter | lunch_counter
Counter({'spam': 3, 'eggs': 2, 'bacon': 1})
```

The item `eggs` is again common to both. Unlike addition, union doesn't add their counts but selects the one with the larger count.

Order by Key with OrderedDict()

This is an example run with the Python 2 interpreter:

```
>>> quotes = {
...     'Moe': 'A wise guy, huh?',
...     'Larry': 'Ow!',
...     'Curly': 'Nyuk nyuk!',
...     }
>>> for stooge in quotes:
...  print(stooge)
...
Larry
Curly
Moe
```

Starting with Python 3.7, dictionaries retain keys in the order in which they are added. `OrderedDict` is useful for earlier versions, which have an unpredictable order. The examples in this section are relevant only if you're using a version of Python earlier than 3.7.

`OrderedDict` remembers the order of key addition and returns the keys in the same order from an iterator. Try creating an `OrderedDict` from a sequence of (*`key`*, *`value`*) tuples:

```
>>> from collections import OrderedDict
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
>>> for stooge in quotes:
...     print(stooge)
...
Moe
Larry
Curly
```

Stack + Queue == deque

Here are definitions of these elements:

- A *stack* is a LIFO data structure, like a stack of trays at a cafeteria.
- A *queue* is a FIFO data structure, like a vending machine, where the item stocked first is the first to tumble down when you buy it. Unless it gets stuck.
- A *deque* (pronounced *deck*) is a double-ended queue, which has features of both a stack and a queue. It's useful when you want to add and delete items from either end of a sequence.

Let's work from both ends of a word to the middle to see whether it's a palindrome. The function `popleft()` removes the leftmost item from the deque and returns it; `pop()` removes the rightmost item and returns it. Together, they work from the ends toward the middle. As long as the end characters match, the function keeps popping until it reaches the middle:

```
>>> def palindrome(word):
...     from collections import deque
...     dq = deque(word)
...     while len(dq) > 1:
...        if dq.popleft() != dq.pop():
...            return False
...     return True
...
...
>>> palindrome('a')
True
>>> palindrome('racecar')
True
>>> palindrome('')
True
>>> palindrome('radar')
True
>>> palindrome('halibut')
False
```

I use this as a simple illustration of deques. If you really want a quick palindrome checker, comparing a string with its reverse would be a lot simpler. Python doesn't have a `reverse()` function for strings, but it does have a way to reverse a string with a slice, as illustrated here:

```
>>> def another_palindrome(word):
...     return word == word[::-1]
...
>>> another_palindrome('radar')
True
>>> another_palindrome('halibut')
False
```

Iterate over Code Structures with itertools

The `itertools` module contains special-purpose iterator functions. Each returns one item at a time when called within a `for` … `in` loop, and remembers its state between calls. Looking at `itertools` is *definitely* worth your time.

The `chain()` function runs through its arguments as though they were a single iterable:

```
>>> import itertools
>>> for item in itertools.chain([1, 2], ['a', 'b']):
```

```
...     print(item)
...
1
2
a
b
```

The `cycle()` function is an infinite iterator, cycling through its arguments:

```
>>> import itertools
>>> for item in itertools.cycle([1, 2]):
...     print(item)
...
1
2
1
2
.
.
.
```

And so on.

The `accumulate()` function calculates accumulated values. By default, it calculates the sum:

```
>>> import itertools
>>> for item in itertools.accumulate([1, 2, 3, 4]):
...     print(item)
...
1
3
6
10
```

You can provide a function as the second argument to `accumulate()`, and it will be used instead of addition. The function should take two arguments and return a single result. This example calculates an accumulated product:

```
>>> import itertools
>>> def multiply(a, b):
...     return a * b
...
>>> for item in itertools.accumulate([1, 2, 3, 4], multiply):
...     print(item)
...
1
2
6
24
```

The `itertools` module has many more functions, notably some for combinations and permutations that can be time savers when the need arises.

Get Random

We played with `random.choice()` at the beginning of this chapter. That returns a value from the sequence (list, tuple, dictionary, string) argument that it's given:

```
>>> from random import choice
>>> choice([23, 9, 46, 'bacon', 0x123abc])
1194684
>>> choice( ('a', 'one', 'and-a', 'two') )
'one'
>>> choice(range(100))
68
>>> choice('alphabet')
'l'
```

Use the `sample()` function to get more than one value at a time:

```
>>> from random import sample
>>> sample([23, 9, 46, 'bacon', 0x123abc], 3)
[1194684, 23, 9]
>>> sample(('a', 'one', 'and-a', 'two'), 2)
['two', 'and-a']
>>> sample(range(100), 4)
[54, 82, 10, 78]
>>> sample('alphabet', 7)
['l', 'e', 'a', 't', 'p', 'a', 'b']
```

To get a random integer from any range, you can use `choice()` or `sample()` with `range()`, or use `randint()` or `randrange()`:

```
>>> from random import randint
>>> randint(38, 74)
71
>>> randint(38, 74)
60
>>> randint(38, 74)
61
```

The `randrange()` function, like `range()`, has arguments for the start (inclusive) and end (exclusive) integers, and an optional integer step:

```
>>> from random import randrange
>>> randrange(38, 74)
65
>>> randrange(38, 74, 10)
68
>>> randrange(38, 74, 10)
48
```

Finally, get a random real number (a float) from 0.0 to 1.0:

```
>>> from random import random
>>> random()
0.07193393312692198
```

```
>>> random()
0.7403243673826271
>>> random()
0.9716517846775018
```

More Batteries: Get Other Python Code

Sometimes the standard library doesn't have what you need or doesn't do it in quite the right way. There's an entire world of open source, third-party Python software. Good resources include the following:

- PyPI (also known as the Cheese Shop, after an old Monty Python skit)
- GitHub
- Read the Docs

You can find many smaller code examples at ActiveState.

Almost all the Python code in this book uses the standard Python installation on your computer, which includes all the built-ins and the standard library. External packages are featured in some places, with details on how to install and use them.

Review/Preview

Modules are the next high-level code structure above functions. They isolate code and data in a *namespace* that qualifies names and allows the same name to be used in different places without confusion.

The next chapter begins Part II: the tools to hone your Python skills.

Practice

12.1 Create a file called *zoo.py*. In it, define a function called `hours()` that prints the string `Open 9-5 daily`. Then use the interactive interpreter to import the `zoo` module and call its `hours()` function.

12.2 In the interactive interpreter, import the `zoo` module as `menagerie` and call its `hours()` function.

12.3 Staying in the interpreter, import the `hours()` function from `zoo` directly and call it.

12.4 Import the `hours()` function as `info` and call it.

12.5 Make a dictionary called `plain` with the key-value pairs `'a': 1`, `'b': 2`, and `'c': 3`, and then print it.

12.6 Make an `OrderedDict` called `fancy` from the same pairs listed in the previous question and print it. Did it print in the same order as `plain`?

12.7 Make a `defaultdict` called `dict_of_lists` and pass it the argument `list`. Make the list `dict_of_lists['a']` and append the value `something for a` to it in one assignment. Print `dict_of_lists['a']`.

PART II
Tools

Part I wandered the vaults of Castle Python, laying out the basics of the Python programming language. In Part II, we'll explore how to become a real, true, knightly Python developer—a *Pythonista*. What tools will you use every day, and which ones only in dire need? We'll avoid those that are positively medieval (like those in the preceding image), and emphasize those that are relatively modern, lightweight, and presenting minimal danger to your extremities. These include the following:

- Setting up your development environment (Chapter 13)
- Type hints and documentation (Chapter 14)
- Testing (Chapter 15)
- Debugging (Chapter 16)

More is sprinkled through Part III later, and we'll get there.

CHAPTER 13
Development Environment

> *Always wanted to travel back in time to try fighting a younger version of yourself? Software development is the career for you!*
>
> —Elliot Loh

Part I of this book mostly described the syntax of the Python programming language. The only computer tools you have really needed so far have been these:

- A text editor, to write the Python code
- A terminal, to run those Python programs

This chapter is devoted to the art and science of Python development, with "best practice" recommendations including other tools, concepts, and ways of organizing your work. Absorb them, and you too can be a card-carrying Pythonista:

- How to find, install, and update Python code
- IDEs—tools beyond the basic text editor and Terminal
- Source control using Git

Find Python Code

When you need to develop code, the fastest solution is to steal it—from a source that allows it.

The Python standard library is wide, deep, and mostly clear. Everything in it is free to use. Dive in and look for those pearls.

Like the halls of fame for various sports, it takes time for a module to get into the standard library. New packages are constantly appearing; throughout this book, I've

highlighted some that either do something new or do something old better. Python is advertised as *batteries included*, but you might need a new kind of battery.

So where, outside the standard library, should you look for good Python code?

- The first place to look is the Python Package Index (PyPI). Formerly named the *Cheese Shop* after a Monty Python skit, this site is constantly updated with Python packages—over 600,000 as I write this. When you use `pip` (see the next section), it searches PyPI. The main PyPI page shows the most recently added packages. You can also conduct a direct search by typing something into the search box in the middle of the PyPI home page. For example, `genealogy` yields 69 matches, and `movies` yields 1,814.
- Another popular repository is GitHub. See all Python packages there (ordered by GitHub *stars*, which indicate popularity) or those that are currently trending.
- "Popular Python Recipes" has more than four thousand short Python programs, on every subject.

Install Packages

There are many ways to install Python packages:

- Use pip if you can. It's the most common method by far. You can install most of the Python packages you're likely to encounter with pip.
- Use Pipenv, which combines pip and virtualenv.
- Or try Poetry, which tries to improve on Pipenv.
- If you work a lot with scientific Python packages, Conda is worth a look.
- Hey, one more: uv is an even newer tool that aims to outdo all the others.

Let's take a look at all of these.

Use Pip

Python packaging has had limitations. An earlier installation tool called `easy_install` was replaced by `pip`, written by Ian Bicking.

Neither had been in the standard Python installation. So, if you're supposed to install packages by using `pip`, where do you get `pip`? Starting with Python 3.4, `pip` will finally be included with the rest of Python to avoid such existential crises. If you're using an earlier version of Python 3 and don't have `pip`, you can get it via the `pip` documentation site.

The simplest use of `pip` is to install the latest version of a single package by using the following command: `pip install flask`.

You will see details on what it's doing, just so you don't think it's goofing off: downloading, running *setup.py*, installing files on your disk, and other details.

You can also ask `pip` to install a specific version: `pip install flask==0.9.0`.

Or a minimum version (this is useful when a feature that you can't live without turns up in a particular version):

```
$ pip install "flask>=0.9.0"
```

In the preceding example, those double quotes prevent the `>=` from being interpreted by the shell to redirect output to a file called *=0.9.0*.

If you want to install more than one Python package, you can use a requirements file. Although it has many options, the simplest use is a list of packages, one per line, optionally with a specific or relative version:

```
$ pip install -r requirements.txt
```

Your sample *requirements.txt* file might contain this:

```
flask==0.9.0
django
psycopg2
```

Here are two more examples:

- Install the latest version: `pip install --upgrade` *package*
- Delete a package: `pip uninstall` *package*

Install with a Native Package Manager

Apple's macOS includes the third-party packagers Homebrew (`brew`) and MacPorts (`ports`). They work a little like `pip` but aren't restricted to Python packages.

Linux has a different manager for each distribution. The most popular are `apt-get`, `yum`, `dpkg`, and `zypper`.

Windows has the Windows Installer and package files with a *.msi* suffix. If you installed Python for Windows, it was probably in the MSI format.

Install from Source

Occasionally, a Python package is new, or the author hasn't managed to make it available with `pip`. To build the package, you generally do the following:

1. Download the code.
2. Extract the files by using `zip`, `tar`, or another appropriate tool if they're archived or compressed.
3. Run `python setup.py install` in the directory containing a *setup.py* file.

As always, be careful what you download and install. It's a little harder to hide malware in pure Python programs, which are readable text instead of binary files, but it has happened.

Virtual Environments

In Python development, it's common to use different versions of the Python interpreter for different projects, and different versions of many modules. How do you control which one you're using at any particular time?

Operating systems have the concept of a *path* for applications. When you type a command, the system looks in a set of directories (folders) until it finds a file that matches that command name. Usually, your current directory comes first, then some standard system directories, but you can change the path. In Unix-like systems, the environment variable `$PATH` is a colon-separated string of directory names.

Environment variables are a set of strings saved by your shell program and provided to every program that it executes. In Linux and macOS, the command `env` shows all your current environment variables. In Windows, use `set`.

You don't want to install software into those system directories because of the following:

- You don't have permission to write to them.
- Even if you could, overwriting your system's standard programs (like `python`) could cause problems.

Python is more dynamic than many other languages, which makes setting and resetting the path awkward. The standard solution is to manage a *virtual environment*—a

directory that contains the Python interpreter, other programs like `pip`, and some packages.

Creating a virtual environment involves these steps:

1. Create a new directory.
2. Install Python and some standard packages into that directory.
3. *Activate* the new environment: get your shell to use this new directory.

Virtualenv and Venv

The standard ways of installing a virtual environment are the third-party package virtualenv or the standard Python module `venv`. Let's first make a virtual environment called `mordor` (a virtual environment of another sort, if there ever was one):

```
$ python -m venv mordor
$ source mordor/bin/activate
$ which python
.../mordor/bin/python
```

(The `...` stands for the directory I was in when I ran the first command, and that directory will be different for you.)

The first command created the subdirectory *mordor* and installed `python` and some modules.[1] The second command put `.../mordor/bin` in your `$PATH` variable to make your shell program look there first for the main `python` program and modules. The third command checks if this worked.

Now, when you type **`python file.py`**, your shell will look under the new `mordor` virtual environment for the Python file *file.py*. Also, any new modules that you install with `pip` will be placed there too.

All of this applies only to the shell in the terminal window that you're using. Other windows run their own shells and are not affected by what you do in this one. They're safe, and your system directories stay unmodified.

A similar third-party program, which like pip was created by Ian Bicking, is called `virtualenv`. First, use `pip install virtualenv`. The process to create a virtual environment is almost the same as before:

```
$ virtualenv mordor
$ source mordor/bin/activate
$ which python
.../mordor/bin/python
```

1 Or linked to versions that were already on disk elsewhere.

Pipenv

In practice, Python developers use `pip` and virtual environments a lot. Naturally, some people have tried to combine them to save a few keystrokes and brain cells.

A package called Pipenv combines our friends `pip` and `virtualenv`. It also addresses dependency issues that can arise when using `pip` in different environments (such as your local development machine, versus staging, versus production). Install Pipenv with `pip install pipenv`.

Poetry

Similar to Pipenv, Poetry manages Python packages and their dependencies. It requires Python version 3.9 or higher. Install Poetry with `pip install poetry`. It depends on a *pyproject.toml* file. See Poetry's documentation for the details.

Conda

Conda is a system package manager. It can handle other things besides Python and is geared toward scientific software development.

uv

The most modern of the popular Python package managers, uv is written in Rust and is gaining popularity quickly. It's much faster than its competitors and is compared with them in the articles "Poetry versus uv", "Revisiting uv", and "Python UV: The Ultimate Guide to the Fastest Python Package Manager.

This is not the same as libuv, a "cross-platform library for asynchronous I/O."

If you're willing to try something new, uv might replace all the tools that I just described in the previous sections.

Integrated Development Environments

I started programming in the ancient green text console days,[2] and got used to minimal development environments. So far, I've assumed that all you've needed to try the programs in this book are a text window and editor in your computer's graphic operating system. But you can do a lot more with many available IDEs:

- Code editing
- Autocomplete
- Documentation
- Formatting
- Style checking
- Filesystem interface
- Source management
- Testing
- Debugging

Let's look at some of them—text-based ones first, then those with graphic interfaces. You can read more detailed comparisons from the sites Real Python, simplilearn, and Full Stack Python.

These are some text-based development tools:

Notepad++
: A Windows stalwart.

vi
: A standard editor in Unix-like systems for decades.

Vim
: Has largely supplanted the original vi.

Neovim
: Written to make Vim's code more sustainable and adaptable. Although it's a text-based system at its core, Neovim has been adapted to GUI-based IDEs like Visual Studio Code.

2 Visualize a grizzled prospector (long beard, few teeth) drawling, "Gee whillikers, all we had in them days was zeros and ones, and we was durned glad to have them ones."

Let's move up to some souped-up code editors:

Sublime Text
: A step up from the pure vi-family editors.

IDLE
: Python's *Integrated Development and Learning Environment* (and a sly nod to Eric Idle of Monty Python) is the only Python IDE that's included with the standard distribution.

Zed
: A new multilanguage IDE, written in Rust for speed, that connects with AI large language models (LLMs).

If you watch Python videos on YouTube, the presenters are almost always using a GUI environment, zipping around on the screen to select files and run chunks of code. The most popular graphic IDEs include the following:

PyCharm
: A recent graphic IDE with many features. The Community Edition is free, and you can get a free license for the Professional Edition to use in a classroom or an open source project.

Visual Studio (VS) Code
: Free and extremely popular. Download the core product and at least the Python extension.

Cursor
: A recent AI-based fork (derived version) of VS Code.

IPython

IPython started as an enhanced terminal (text) Python IDE but evolved a graphical interface that looks like a notebook. It integrated many packages that are discussed in this book, including Matplotlib and NumPy, and became a popular tool in scientific computing.

You install the basic text version with (you guessed it) `pip install ipython`. When you start it, you'll see something like this:

```
$ ipython
Python 3.13.0 (v3.13.0:60403a5409f, Oct  7 2024, 00:37:40)
[Clang 15.0.0 (clang-1500.3.9.4)]
Type copyright, credits or license for more information
IPython 9.2.0 -- An enhanced Interactive Python. Type ? for help.
Tip: Use `F2` or %edit with no arguments to open an empty editor with ...

In [1]:
```

As you know, the standard Python interpreter uses the input prompts `>>>` and `...` to indicate where and when you should type code. IPython tracks everything you type in a list called `In`, and all your output in `Out`. Each input can be more than one line, so you submit it by holding the Shift key while pressing Enter.

Here's a one-line example:

```
In [1]: print("Hello? World?")
Hello? World?

In [2]:
```

`In` and `Out` are automatically numbered lists, letting you access any of the inputs you typed or outputs you received.

If you type `?` after a variable, IPython tells you its type, value, ways of making a variable of that type, and an explanation:

```
In [4]: answer = 42

In [5]: answer?
Type:       int
String Form:42
Docstring:
int(x=0) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

Name lookup is a popular feature of IDEs such as IPython. If you press the Tab key right after some characters, IPython shows all variables, keywords, and functions that begin with those characters (as does the standard Python interpreter). Let's define some variables and then find everything that begins with the letter `f`:

```
In [6]: fee = 1

In [7]: fie = 2

In [8]: fo = 3

In [9]: fum = 4
```

```
In [10]: ftab
%%file    fie       finally   fo        format    frozenset
fee       filter    float     for       from      fum
```

If you type `fe` followed by the Tab key, your input expands to the variable `fee`, which, in this program, is the only word that starts with `fe`:

```
In [11]: fee
Out[11]: 1
```

There's much more to IPython. Take a look at its tutorial to get a feel for its features.

Jupyter Notebook

Jupyter is an evolution of IPython. The name combines the languages Julia, Python, and R—all of which are popular in data science and scientific computing. Jupyter Notebook is a modern way to develop and publish your code with documentation for any of these languages.

If you'd like to play with Jupyter Notebook first before installing anything on your computer, you can first try it out in your web browser. To install Jupyter Notebook locally, type `pip install jupyter`. Run it with `jupyter notebook`.

JupyterLab

JupyterLab is the next generation of Jupyter Notebook and will eventually replace it. As with Notebook, you can first try out JupyterLab in your browser. You install it locally with `pip install jupyterlab` and then run it with `jupyter lab`.

Source Control

When you're working on a small group of programs, you can usually keep track of your changes—until you make a harebrained mistake and clobber a few days of work. Source-control systems help protect your code from mistakes. If you work with a group of developers, source control becomes a necessity.

Many commercial and open source packages are available in this area. The most popular in the open source world where Python lives are Mercurial and Git. Both are examples of *distributed* version control systems, which produce multiple copies of code repositories. Earlier systems, such as Subversion, run on a single server.

Mercurial

Mercurial is written in Python. It's fairly easy to learn, with a handful of subcommands to download code from a Mercurial repository, add files, check in changes, and merge changes from different sources. Bitbucket and other sites offer free or commercial hosting.

Git

Git was originally written by Linus Torvalds for Linux kernel development but now dominates open source in general. Git is similar to Mercurial, although some find it slightly trickier to master. GitHub is the largest Git host, with more than a million repositories.

This book was authored with an internal O'Reilly product called Atlas, and its original text and images were hosted on a private Git repository.

The standalone program examples in this book are available in a public Git repository at GitHub. If you have the `git` program on your computer, you can download these programs by using this command: `git clone https://github.com/madscheme/introducing-python`.

You can also download the code from the GitHub page:

- Click "Clone in Desktop" to open your computer's version of Git, if installed.
- Click Download ZIP to get a zipped archive of the programs.

If you don't have Git but would like to try it, read the installation guide. I talk about the command-line version here, but you might be interested in sites such as GitHub that have extra services and might be easier to use in some cases; Git has many features but is not always intuitive.

Let's take Git for a test drive. We won't go far, but the ride will show a few commands and their output.

Make a new directory and change to it:

```
$ mkdir newdir
$ cd newdir
```

Create a local Git repository in your current directory *newdir*:

```
$ git init
Initialized empty Git repository in .../newdir/.git/
```

In *newdir*, create a Python file called *test.py*, as shown in Example 13-1.

Example 13-1. test.py

```
print('Oops')
```

Add the file to the Git repository:

```
$ git add test.py
```

What do you think of that, Mr. Git?

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   test.py
```

This means that *test.py* is part of the local repository, but its changes have not yet been committed. Let's *commit* the file:

```
$ git commit -m "simple print program"
 [master (root-commit) 52d60d7] my first commit
  1 file changed, 1 insertion(+)
  create mode 100644 test.py
```

That `-m "my first commit"` is your *commit message*. If you omit that, `git` will pop you into an editor and coax you to enter the message that way. This becomes a part of the `git` change history for that file.

Let's see our current status:

```
$ git status
 On branch master
 nothing to commit, working directory clean
```

OK, all current changes have been committed. This means that we can make changes and not worry about losing the original version. Make an adjustment now to *test.py*—change `Oops` to `Ops!` and save the file (Example 13-2).

Example 13-2. test.py, revised

```
print('Ops!')
```

Let's check to see what `git` thinks now:

```
$ git status
On branch master

No commits yet
```

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.py
```

Use `git diff` to see which lines have changed since the last commit:

```
$ git diff
diff --git a/test.py b/test.py
index 76b8c39..62782b2 100644
--- a/test.py
+ b/test.py
@@ -1 +1 @@
-print(Oops)
+print(Ops!)
```

If you try to commit this change now, `git` complains:

```
$ git commit -m "change the print string"
On branch master
Changes not staged for commit:
    modified:   test.py

no changes added to commit
```

That `staged for commit` phrase means you need to `add` the file, which roughly translated means *Hey Git, look over here*:

```
$ git add test.py
```

You could have also typed `git add .` to add *all* changed files in the current directory; that's handy when you have edited multiple files and want to ensure that you check in all their changes. Now we can commit the change:

```
$ git commit -m "my first change"
[master 9f6f645] my first change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

If you'd like to see all the terrible things that you've done to *test.py*, most recent first, use `git log` (I've edited the `Author` lines that follow for privacy reasons):

```
$ git log test.py
commit 9f6f645eb353efcb574bd5997c1b064d66d78f75 (HEAD -> master)
Author: name < email >
Date:   Sat May 24 00:27:33 2025 -0500

    my first change

commit e048e3e19e714d7da8c7668dc4f0649045da4a2d
Author: name < email >
Date:   Sat May 24 00:24:41 2025 -0500

    simple print program
```

Review/Preview

Python's ecosystem is wide and deep, so setting up a development environment is a bit more work than just compiling and running a single source file. Useful components include the following:

- A *virtual environment* to manage where your computer looks for Python packages
- A *package and dependency manager* to install those Python packages in the right place, and ensure that versions are matched to work together
- An *IDE* for text editors, source control, documentation access, testing, and other necessities

Coming up: type hints are ignored by the Python interpreter, so why would you care about them? They help document your code, can be used to check for errors, and are required by tools like the web framework FastAPI (see "FastAPI" on page 477).

Practice

13.1 Install `pip` if you don't have it. Make `pip` install the pyspellchecker package.

13.2 Read the pyspellchecker docs at the PyPI and Read the Docs websites. Write a small script that takes a list of words, reports whether any are misspelled, and reports the likely true spelling. Note that more than 10 languages are supported.

13.3 (Optional) Install uv. Use it to check the correctness of the code you wrote for 13.2.

13.4 (Optional) Install PyCharm and VS Code. Try them out.

CHAPTER 14

Type Hints and Documentation

One can never produce anything as terrible and impressive as one can awesomely hint about.
—H. P. Lovecraft

Python was designed so that variables were really just references to objects: names, or sticky notes, attached to the data that contained all the details. So for years, there was little need to say, for instance, what data type a variable referred to. But sometimes we imperfect humans would misspell names, or forget what we were trying to do in long blocks of code. So, eventually, optional *type hints* were added to the language. You use these to indicate what you mean. Type hints are not enforced by the Python interpreter but can serve as documentation, and tools like Mypy understand them and use them for error checking.

Type Hints

Static languages require you to define the types of your variables, and they can catch some errors at compile time. As you know, Python doesn't do this, and you may find bugs only when the code is run. An *object* has a strict type and location in memory, but a *variable* is just a name that can point to any object at any time.

However, in real code (in Python and other languages), a name tends to refer to a specific object. It would help, at least in documentation, if we could annotate code elements (variables, function returns, and so on) with the object types we expect them to be referencing. Then developers would not need to look through as much code to see how a particular variable is supposed to act.

Python 3.5 added *type hints* (or *type annotations*) to address this. Their use is completely optional, and does not force types on variables. Type hints help developers who are used to static languages, where variable types *must* be declared (or can be inferred somehow). These are only hints, and they don't change the way Python

works. Their main use is for documentation, but people are finding more uses. For example, the FastAPI web framework uses hints to generate web documentation with live forms for testing.

You can apply type hints to variables, functions, and object attributes and methods.

Variable Hints

A type hint for a variable might specify only the type:

```
name : type
```

or the type and a value:

```
name : type = value
```

The *type* can be a standard Python type name like `int` or `str`:

```
palindrome: str = "Was it a car or a cat I saw?"
```

For collection types like list, tuple, and dict, you can specify which types are in the collection:

```
name: dict[keytype, valtype] = {key1: val1, key2: val2}
```

```
>>> nays: dict[str, str] = {
... "horse": "neigh",
... "pedant": "nay!",
... "genealogist": "n\N{LATIN SMALL LETTER E WITH ACUTE}e"
... }
>>>
```

If the variable could *really* be of any type (this is uncommon), import `Any` from `typing` and use it:

```
mysterious_thing: Any
```

If the variable could be one of multiple types, you can say this in Python 3.10 and up:

```
number: int | float
```

In earlier versions of Python, you'll need `Union` from the `typing` module:

```
>>> from typing import Union
>>> number: Union(int, float)
```

If a variable might have a type and value, but also might be missing, use `None` (Python 3.10+) or `Optional`:

```
>>> from typing import Optional
>>> quantum_value: float | None = None
>>> quantum_value: Optional[float] = None
```

Function Hints

Hints apply to function parameters and return values. Parameters are treated like the variables in the previous section:

```
def num_to_str(num: int) -> str:
    return str(num)
```

If the function doesn't return anything, use `None`:

```
def no_return(num: int) -> None:
    print("Hey, got a", num)
```

Mypy

Although type hints are ignored by the Python interpreter, *static type checking* tools like Mypy understand them, and complain if a type hint is being violated. Use `pip install mypy` to get the tool.

Let's make a correct test file, then add some type errors and see how alert Mypy is. Call this *test.py* (Example 14-1).

Example 14-1. test.py

```
def int_to_str(num: int) -> str:
    return str(num)

def str_to_int(txt: str) -> int:
    return int(txt)
```

Run Mypy against this:

```
$ mypy test.py
Success: no issues found in 1 source file
```

Now let's put on our gray hats and introduce some errors to *test.py* (Example 14-2).

Example 14-2. test.py with errors

```
def int_to_str(num: int) -> str:
    return str(num)

def str_to_int(txt: str) -> int:
    return int(txt)
```

```
txt = int_to_str(19)
txt = int_to_str(47.5)
txt = int_to_str("")

num = str_to_int("56")
num = str_to_int("")
num = str_to_int(98)
```

Running *test.py* through plain old Python will cause some exceptions (I've shortened some lines here):

```
$ python test.py
Traceback (most recent call last):
  File ".../test.py",
  line 12, in <module>
    num = str_to_int("")
  File ".../test.py",
  line 5, in str_to_int
    return int(txt)
ValueError: invalid literal for int() with base 10: ''
```

But Mypy understands both type hints and Python, and warns us:

```
$ mypy test.py
test.py:8: error: Argument 1 to "int_to_str" has
  incompatible type "float"; expected "int"  [arg-type]
test.py:9: error: Argument 1 to "int_to_str" has
  incompatible type "str"; expected "int"  [arg-type]
test.py:13: error: Argument 1 to "str_to_int" has
  incompatible type "int"; expected "str"  [arg-type]
Found 3 errors in 1 file (checked 1 source file)
```

Mypy checks all the code without running it fully like the Python interpreter, and it catches lines that would have raised exceptions at runtime. It does *not* catch the call to `str_to_int()`, which expects a string of digit characters, and we pass it an empty string. We tell Mypy only that `txt` is a string, not that it has to contain at least one digit character.

Documentation

> *The guy who knows about computers is the last person you want to have creating documentation for people who don't understand computers.*
>
> —Adam Osborne

The code does one thing, and what we say it should do is sometimes another. When the code doesn't speak for itself, there are ways to help. Type hints help in some ways, but explanatory text can say much more.

Comments

I mentioned code comments in "Comment with #" on page 81. These are always recommended near code that might be confusing or obscure to the reader.

Before type hints, people often used *structured comments* to document Python variables and functions. These were often homegrown by each group of developers, but some involved tools like Sphinx or Doxygen.

Docstrings

I mentioned docstrings (an initial string within a function body) in "Docstrings" on page 162. The docstring of a function named `func()` is available as `func.__doc__`.

The Python `help()` function returns the docstring of an object:

```
>>> def echo(something):
...     "echo() returns its single input argument"
...     return something
...
>>> help(echo)

Help on function echo in module __main__:

echo(something)
    echo() returns its single input argument
 Help on echo line 1/4 (END) (press h for help or q to quit)
```

Markup Text Files

Many markup text formats have emerged to express the structure and formatted appearance of documents. HTML is prominent, but you'll see other formats that try to also be more directly readable; examples are Markdown and AsciiDoc.[1] It's common to see markup documentation files alongside Python code files in source repositories; you'll often see a file with a name like *README.md* instead of *index.html*.

Review/Preview

Documentation can take many forms, including code comments, external text files, and Python type hints.

A type hint indicates what your variable is supposed to mean: what internal object data type does it refer to? The Python interpreter doesn't care (as long as it's in a legal format), but tools like Mypy use these hints to catch errors. Also, some modern packages (see "FastAPI" on page 477) require the use of hints.

1 The format of this book's text.

Coming up: testing. It's not always fun, but it's necessary and can save you a lot of grief.

Practice

14.1 Write a function called `pointless()` that accepts a dict of *`string:int`* elements, capitalizes the key, adds 1 to the value, and prints both. Add complete type hints.

CHAPTER 15
Testing

Program testing can be used to show the presence of bugs, but never to show their absence!
—Edsger Dijkstra

You probably already know this: even trivial code changes can break your program. Python lacks the type-checking of static languages, which makes some things easier but also lets undesirable results through the door. Testing is essential.

The simplest way to test Python programs is to add `print()` statements. The Python interactive interpreter's Read-Evaluate-Print Loop (REPL) lets you edit and test changes quickly. However, you don't want `print()` statements in production code, so you need to remember to remove them all.

Pylint

The next step, before creating test programs, is to run a Python code checker. Traditionally, the most popular have been Pylint and Pyflakes. You can install either or both by using `pip`:

```
$ pip install pylint
$ pip install pyflakes
```

These check for code errors (such as referring to a variable before assigning it a value) and style faux pas (the code equivalent of wearing plaids and stripes). Example 15-1 is a fairly meaningless program with a bug and style issue.

Example 15-1. style1.py

```
a = 1
b = 2
print(a)
print(b)
print(c)
```

Here's the initial output of Pylint:

```
$ pylint style1.py
No config file found, using default configuration
************* Module style1
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
E:  5,6: Undefined variable 'c'
```

Much further down, under `Global evaluation`, is our score (10.0 is perfect):

```
Your code has been rated at -3.33/10
```

Ouch. Let's fix the bug first. That Pylint output line starting with an `E` indicates an `Error`, which occurred because we didn't assign a value to `c` before we printed it. Take a look at Example 15-2 to see how we can fix that.

Example 15-2. style2.py

```
a = 1
b = 2
c = 3
print(a)
print(b)
print(c)
```

```
$ pylint style2.py
```

```
No config file found, using default configuration
************* Module style2
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  3,0: Invalid name "c" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
```

Good, no more `E` lines. And our score jumps from –3.33 to 4.29:

```
Your code has been rated at 4.29/10
```

Pylint wants a docstring (a short text at the top of a module or function, describing the code), and it thinks short variable names such as `a`, `b`, and `c` are tacky. Let's make Pylint happier and improve *style2.py* to *style3.py* (Example 15-3).

Example 15-3. style3.py

```
"Module docstring goes here"

def func():
    "Function docstring goes here. Hi, Mom!"
    first = 1
    second = 2
    third = 3
    print(first)
    print(second)
    print(third)

func()
```

```
    $ pylint style3.py
    No config file found, using default configuration
```

Hey, no complaints. And our score?

```
    Your code has been rated at 10.00/10
```

And there was much rejoicing.

Another style checker is pep8, which you can install in your sleep by now:

```
    $ pip install pep8
```

What does it say about our style makeover?

```
    $ pep8 style3.py
    style3.py:3:1: E302 expected 2 blank lines, found 1
```

To be really stylish, it's recommending that I add a blank line after the initial module docstring.

Ruff

There's a new kid in town. Actually, it's a family: tools written in Rust that combine features of existing Python tools and are much faster. Ruff is one of these: a linter and code formatter. Get it with arcane hand gestures and incantations, or just `pip install ruff`.

Let's use Ruff to check the first style example from the previous section:

```
$ ruff check style1.py
style1.py:5:7: F821 Undefined name `c`
  |
3 | print(a)
4 | print(b)
5 | print(c)
  |       ^ F821
  |

Found 1 error.
```

Ruff does *much* more than this teeny example and is well worth a look.

Unittest

We've verified that we're no longer insulting the style senses of the code gods, so let's move on to actual tests of the logic in your program.

It's a good practice to write independent test programs first, to ensure that they all pass before you commit your code to any source-control system. Writing tests can seem tedious at first, but they really do help you find problems faster—especially *regressions* (breaking something that used to work). Painful experience teaches all developers that even the teeniest change, which they swear could not possibly affect anything else, actually does. If you look at well-written Python packages, they always include a test suite.

The standard library contains not one, but two, test packages. Let's start with unittest. We'll write a module that capitalizes words. Our first version uses the standard string function `capitalize()` with some unexpected results, as we'll see. Save this as *cap.py* (Example 15-4).

Example 15-4. cap.py

```
def just_do_it(text):
    return text.capitalize()
```

The basis of testing is to decide the outcome you want from a certain input (here, you want the capitalized version of whatever text you input), submit the input to the function you're testing, and then check whether it returned the expected results. The expected result is called an *assertion*, so in unittest you check your results by using methods with names that begin with `assert`, like the `assertEqual()` method shown in Example 15-5.

Save this test script as *test_cap.py*.

Example 15-5. test_cap.py

```
import unittest
import cap

class TestCap(unittest.TestCase):

    def setUp(self):
        pass

    def tearDown(self):
        pass

    def test_one_word(self):
        text = 'duck'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'Duck')

    def test_multiple_words(self):
        text = 'a veritable flock of ducks'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'A Veritable Flock Of Ducks')


if __name__ == '__main__':
    unittest.main()
```

The `setUp()` method is called before each test method, and the `tearDown()` method is called after each. Their purpose is to allocate and free external resources needed by the tests, such as a database connection or some test data. In this case, our tests are self-contained, and we wouldn't even need to define `setUp()` and `tearDown()`, but it doesn't hurt to have empty versions there. At the heart of our test are the two functions named `test_one_word()` and `test_multiple_words()`. Each runs the `just_do_it()` function we defined with different inputs and checks whether we got back what we expected. OK, let's run *test_cap.py*. This will call our two test methods:

```
$ python test_cap.py
F.
======================================================================
FAIL: test_multiple_words (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cap.py", line 20, in test_multiple_words
 self.assertEqual(result, 'A Veritable Flock Of Ducks')
AssertionError: 'A veritable flock of ducks' != 'A Veritable Flock Of Ducks'
- A veritable flock of ducks
?   ^         ^     ^  ^
+ A Veritable Flock Of Ducks
?   ^         ^     ^  ^


----------------------------------------------------------------------
```

```
Ran 2 tests in 0.001s

FAILED (failures=1)
```

The first test (`test_one_word`) succeeded, but the second (`test_multiple_words`) failed. The up arrows (^) show where the strings differed.

What's special about multiple words? Reading the documentation for the `string capitalize()` function yields an important clue: it capitalizes only the first letter of the first word. Maybe we should have read that first.

Consequently, we need another function. Gazing down that page a bit, we find `title()`. So, let's change *cap.py* to use `title()` instead of `capitalize()` (Example 15-6).

Example 15-6. cap.py, revised

```
def just_do_it(text):
    return text.title()
```

Rerun the tests, and let's see what happens:

```
$ *python test_cap.py(

 ..
 ----------------------------------------------------------------------
 Ran 2 tests in 0.000s

 OK
```

Everything is great. Well, actually not. We need to add at least one more method to *test_cap.py* (Example 15-7).

Example 15-7. test_cap.py, revised

```
    def test_words_with_apostrophes(self):
        text = "I'm fresh out of ideas"
        result = cap.just_do_it(text)
        self.assertEqual(result, "I'm Fresh Out Of Ideas")
```

Go ahead and try the code again:

```
$ python test_cap.py

 ..F
 ======================================================================
 FAIL: test_words_with_apostrophes (__main__.TestCap)
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "test_cap.py", line 25, in test_words_with_apostrophes
     self.assertEqual(result, "I'm Fresh Out Of Ideas")
```

```
AssertionError: "I'M Fresh Out Of Ideas" != "I'm Fresh Out Of Ideas"
- I'M Fresh Out Of Ideas
?   ^
+ I'm Fresh Out Of Ideas
?   ^


----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

Our function capitalizes the `m` in `I'm`. A quick run back to the documentation for `title()` shows that it doesn't handle apostrophes well. We *really* should have read the entire text first.

At the bottom of the standard library's `string` documentation is another candidate: a helper function called `capwords()`. Let's use it in *cap.py* (Example 15-8).

Example 15-8. cap.py, re-revised

```
def just_do_it(text):
    return capwords(text)
```

```
$ python test_cap.py

...
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
```

At last, we're finally done! Uh, no. We have one more test to add to *test_cap.py* (Example 15-9).

Example 15-9. test_cap.py, re-revised

```
    def test_words_with_quotes(self):
        text = "\"You're despicable,\" said Daffy Duck"
        result = cap.just_do_it(text)
        self.assertEqual(result, "\"You're Despicable,\" Said Daffy Duck")
```

Does it work?

```
$ python test_cap.py

...F
======================================================================
FAIL: test_words_with_quotes (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
  File "test_cap.py", line 30, in test_words_with_quotes
    self.assertEqual(result, "\"You're
    Despicable,\" Said Daffy Duck") AssertionError: '"you\'re Despicable,"
        Said Daffy Duck'
 != '"You\'re Despicable," Said Daffy Duck' - "you're Despicable,"
        Said Daffy Duck
?  ^ + "You're Despicable," Said Daffy Duck
?  ^
----------------------------------------------------------------------
Ran 4 tests in 0.004s

FAILED (failures=1)
```

It looks like that first double quote confuses even `capwords()`, our favorite capitalizer thus far. It tries to capitalize the ", and lowercases the rest (`You're`). We should have also tested that our capitalizer left the rest of the string untouched.

People who do testing for a living have a knack for spotting these edge cases, but developers often have blind spots when it comes to their own code.

The unittest package provides a small but powerful set of assertions, letting you check values, confirm whether you have the class you want, determine whether an error was raised, and so on.

Doctest

The second test package in the standard library is doctest. With this package, you can write tests within the docstring itself, also serving as documentation. It looks like the interactive interpreter: the characters `>>>`, followed by the call, and then the results on the following line. You can run some tests in the interactive interpreter and just paste the results into your test file. Let's modify our old *cap.py* as *cap2.py* (without that troublesome last test with quotes), as shown in Example 15-10.

Example 15-10. cap2.py

```
def just_do_it(text):
    """
    >>> just_do_it('duck')
    'Duck'
    >>> just_do_it('a veritable flock of ducks')
    'A Veritable Flock Of Ducks'
    >>> just_do_it("I'm fresh out of ideas")
    "I'm Fresh Out Of Ideas"
    """
    from string import capwords
    return capwords(text)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

When you run *cap2.py*, it doesn't print anything if all tests passed:

```
$ python cap2.py
```

Give the command the verbose (`-v`) option to see what actually happened:

```
$ python cap2.py -v
Trying:
    just_do_it(duck)
    Expecting:
    Duck
ok
Trying:
     just_do_it(a veritable flock of ducks)
Expecting:
     A Veritable Flock Of Ducks
ok
Trying:
     just_do_it("I'm fresh out of ideas")
Expecting:
     "I'm Fresh Out Of Ideas"
ok
1 items had no tests:
     main
1 items passed all tests:
    3 tests in main.just_do_it
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

Pytest

One of the most popular Python test tools, if not the most popular, is pytest. Install it with `pip install pytest`. Its features include the following:

Test discovery
: You can tell pytest which test files to run. If you don't provide any filenames, any function and Python filename with the phrase *test* in its prefix or suffix, in or below the current directory, will be run automatically.

Test failure details
: Any failing `assert` statements in the tests print what was expected and what happened, instead of just printing exceptions.

Fixtures
: These are functions that provide parameters like test data, or initialize details like database connections. Each fixture has a *scope* that sets how often it's run, such as for every test or just once per test file.

Parameterization
: These provide multiple pieces of data to a test function.

Examples

You write one or more Python scripts with *test* in the prefix or suffix, and pytest runs them all if you don't specify which files to run. Each script should contain test functions with `assert` statements that should be true.

Let's redo the capitalization example from "Unittest" on page 258. We no longer need `setUp()` and `tearDown()` functions.

We'll keep our original *cap.py* code from Example 15-8 for Example 15-11.

Example 15-11. cap.py

```
from string import capwords

def just_do_it(text):
    return capwords(text)
```

The test functions are a lot shorter and don't involve any classes (Example 15-12).

Example 15-12. test_cap_pytest.py

```
import cap

def test_one_word():
    text = 'duck'
    result = cap.just_do_it(text)
    assert result ==  'Duck'

def test_multiple_words():
    text = 'a veritable flock of ducks'
    result = cap.just_do_it(text)
    assert result == 'A Veritable Flock Of Ducks'

def test_words_with_apostrophes():
    text = "I'm fresh out of ideas"
    result = cap.just_do_it(text)
    assert result == "I'm Fresh Out Of Ideas"

def test_words_with_quotes():
    text = "'You\'re despicable!' said Daffy Duck"
    result = cap.just_do_it(text)
    assert result == "'You\'re Despicable!' Said Daffy Duck"
```

Let's run the tests in pytest's quiet, or terse (`-q`), mode:

```
$ pytest -q test_cap_pytest.py

...F                                                                  [100%]
================================= FAILURES =================================
_____________________________ test_words_with_quotes _____________________________

    def test_words_with_quotes():
        text = "'You\'re despicable!' said Daffy Duck"
        result = cap.just_do_it(text)
>       assert result == "'You\'re Despicable!' Said Daffy Duck"
E       assert "'you're Desp...id Daffy Duck" == "'You're Desp...id Daffy Duck"
E
E         - 'You're Despicable!' Said Daffy Duck
E         ?  ^
E         + 'you're Despicable!' Said Daffy Duck
E         ?  ^

test_cap_pytest.py:21: AssertionError
========================== short test summary info ==========================
FAILED test_cap_pytest.py::test_words_with_quotes - assert
"'you're Desp...id Daffy Duck" == "'You're Desp...id Daffy Duck"
1 failed, 3 passed in 0.27s
```

Those three dots at the top indicate that the first three tests succeeded, but the fourth test got a red `F`. As in our earlier tests, the use of `capwords()` in *cap.py* works for only three of our four tests.

If you run `pytest` with `-v`, you'll get a more verbose, line-by-line output.

Fixtures

Fixtures are pytest's equivalent of the `setUp()` and `tearDown()` helper functions. They provide inputs to the test functions.

Write the fixture function, preceded by the decorator `pytest.fixture`. Use that function name as an argument to your test functions. Pytest will call the function and return its return value as that argument. Example 15-13 shows how to define and call two fixtures.

Example 15-13. fixtures.py

```
import pytest
import datetime
import sqlite3

@pytest.fixture
def curtime():
    # Get the current date and time in UTC
    return datetime.datetime.now(datetime.UTC)
```

```
@pytest.fixture
def dbconn():
    # Initialize a memory-based SQLite database connection
    return sqlite3.connect(":memory:")

def test_something(curtime, dbconn):
    print("In test_something()")
    print("Current time is", curtime)
    print("The database connection is", dbconn)

def test_another(curtime, dbconn):
    print("In test_another()")
    print("Current time is", curtime)
    print("The database connection is", dbconn)
```

Let's run this with pytest (the `-s` makes pytest print to the terminal, because it normally eats the output of any `print()` statements; the `-v` is a bit more verbose than the default output):

```
$ pytest -s fixtures.py
============================ test session starts ==============================
...
collected 2 items

fixtures.py::test_something In test_something()
Current time is 2025-06-01 18:14:33.782791+00:00
The database connection is <sqlite3.Connection object at 0x109f6bc40>
PASSED
fixtures.py::test_another In test_another()
Current time is 2025-06-01 18:14:33.784091+00:00
The database connection is <sqlite3.Connection object at 0x109f6bd30>
PASSED

============================== 2 passed in 1.28s ==============================
```

You may notice that the values of `curtime` and `dbconn` differ in the two tests. By default, a fixture is called anew by every test function that uses its name as an argument. This is called having *function scope*. You probably want that for `curtime`, but tasks that take a while, like database connections, are usually called less often. To make `dbconn()` get called only once for all the tests in this file, use *module scope* or *session scope*, as shown in Example 15-14.

Example 15-14. fixtures2.py

```
import pytest
import datetime
import sqlite3

@pytest.fixture(scope="function")
```

```
def curtime():
    # Get the current date and time in UTC
    return datetime.datetime.now(datetime.UTC)

@pytest.fixture(scope="module")
def dbconn():
    # Initialize a memory-based SQLite database connection
    return sqlite3.connect(":memory:")

def test_something(curtime, dbconn):
    print("In test_something()")
    print("Current time is", curtime)
    print("The database connection is", dbconn)

def test_another(curtime, dbconn):
    print("In test_another()")
    print("Current time is", curtime)
    print("The database connection is", dbconn)
```

Now the `dbconn` value is the same across both tests:

```
$ pytest -s fixtures2.py
============================= test session starts ==============================
...
collected 2 items

fixtures2.py::test_something In test_something()
Current time is 2025-06-01 18:21:28.166182+00:00
The database connection is <sqlite3.Connection object at 0x106a2fc40>
PASSED
fixtures2.py::test_another In test_another()
Current time is 2025-06-01 18:21:28.167477+00:00
The database connection is <sqlite3.Connection object at 0x106a2fc40>
PASSED

============================== 2 passed in 1.19s ===============================
```

The first test function that calls a fixture gets its value. If the scope is longer than `function`, that value is cached and reused for the rest of the scope.

Parametrization

With four test functions but only one test string each, our tests are only a little better than nothing. We can use pytest's parametrization to feed a sequence of arguments to the test functions.

The arguments to the decorator `pytest.mark.parametrize` are as follows:

- A string with the comma-separated names of the arguments
- A list containing a tuple of each group of arguments

The test function has argument names that match that first string in the decorator. In Example 15-15, we have a test that's simpler than a bread sandwich: add two numbers and see whether they match the provided sum.

Example 15-15. param.py

```
import pytest

def addem(one, two):
   return one + two

@pytest.mark.parametrize("one,two,sum",
   [(0,0,0), (0,1,1), (1,1,2), (-1,-1,-2)])
def test_addem(one, two, sum):
   assert addem(one, two) == sum
```

Run the test:

```
$ pytest -q param.py
....                                                                 [100%]
4 passed in 0.18s
```

An alternative would be to include the list of tuples in `test_addem()` itself and then iterate it to pass each tuple to `addem()`.

Hypothesis

An alternative test strategy is *property-based testing*: Hypothesis. Instead of thinking up weird data to ensure that your functions can handle anything, you describe the data characteristics and let the tool generate the test data.

This can include outliers that you might have missed, like `NaN` for a number, or `None` for a string.

Install Hypothesis with the usual flair: `pip install hypothesis`. Here's a simple example of its use. Get the square root of a float and test that it's approximately right:

```
>>> from hypothesis import strategies as s, given as g
>>> from math import sqrt, fabs
>>>
>>> @g(f=s.floats())
... def test_sqrt(f):
...     root = sqrt(f)
...     assert fabs(f - (root*root)) < 1.0
...
>>> test_sqrt()
Traceback (most recent call last):
  File "<python-input-10>", line 1, in <module>
    test_sqrt()
    ~~~~~~~~~^^
```

```
    File "<python-input-9>", line 2, in test_sqrt
      def test_sqrt(f):
                    ^^
    File "/Library/Frameworks/Python.framework/Versions/3.13/.../core.py",
    line 1839, in wrapped_test
      raise the_error_hypothesis_found
    File "<python-input-9>", line 3, in test_sqrt
      root = sqrt(f)
  ValueError: math domain error
  Falsifying example: test_sqrt(
      f=-1.0,
  )
```

By default, Hypothesis generates 100 values from `floats()` as arguments to `test_sqrt()`. It looks like one of them is `-1.0`, which is definitely not nice for a square-root function.

Hypothesis is much richer than this tiny example implies. In particular, a large set of extensions generates data types beyond the built-in Python types. For example, Schemathesis tests websites with REST APIs defined by OpenAPI, such as FastAPI.

Nox

Nox is a standalone program for testing Python projects with multiple Python environments. It's an evolution of tox.

To configure Nox, you edit a Python file named *noxfile.py*. Each function in it can be preceded by the decorator `@nox.session`. Then, whenever you run the Nox program, it looks for these functions in *noxfile.py* and runs them.

Why would you look at Nox? It's a way to manage broader testing of Python projects, including different Python interpreter versions and tool combinations.

Continuous Integration

When your group is cranking out a lot of code daily, it helps to automate tests as soon as changes arrive. You can automate source-control systems to run tests on all code as it's checked in. This way, everyone knows whether someone *broke the build* and just disappeared for an early lunch—or a new job.

These are big systems, and I'm not going into installation and usage details here. In case you need them someday, you'll know where to find them:

Buildbot
: Written in Python, this source-control system automates building, testing, and releasing.

Jenkins
: This is written in Java and seems to be the preferred continuous integration (CI) tool of the moment.

Travis CI
: This automates projects hosted at GitHub and is free for open source projects.

CircleCI
: This one is commercial but free for open source and private projects.

GitHub Actions
: I haven't used this in production, but it's been recommended.

Review/Preview

This was a light overview of Python testing. More targeted tests are in specific areas such as Chapter 24.

You've tested up the wazoo, which was quite an effort for you and the wazoo.[1] But something still went boing/twang/rattle/thud. The next chapter shares the fun of debugging. Expletives are optional.

Practice

15.1 Find out what's wrong with the code in Example 15-16. Use Pylint, Pyflakes, and Ruff (install them if needed).

Example 15-16. errors.py

```
import math

print("square root of 3 is", sqrt(3))
```

15.2 Write a Python file that contains the following:

- A function that adds two integers and returns their sum
- Some pytest tests of this function
- A fixture that returns the current date and time when the file was called

1 Do French developers test up the *oiseau*?

CHAPTER 16
Debugging

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

—Brian Kernighan

Debugging is like being the detective in a crime movie where you are also the murderer.

—Filipe Fortes

If a kid asks where rain comes from, I think a cute thing to tell him is "God is crying." And if he asks why God is crying, another cute thing to tell him is "Probably because of something you did."

—Jack Handey

Test first (Chapter 15). The better your tests are, the less you'll have to fix later. Yet bugs happen and need to be fixed when they're found later.

When code breaks, it's usually because of something you just did. So you typically debug "from the bottom up," starting with your most recent changes.[1]

But sometimes the cause is elsewhere, in something that you trusted and thought worked. You would think that if there were problems in something that many people use, someone would have noticed by now. That is not always what happens. The trickiest bugs that I've encountered, that each took more than a week to fix, were caused by external code that I assumed was good. So *after* blaming the person in the mirror, *question your assumptions*. This is a "top-down" approach, and it takes longer.

1 You, as the detective: "I know I'm in there! And if I don't come out with my hands up, I'm coming in after me!"

Especially insidious bugs can appear when you fix code that was hiding another latent bug.

In Python, unlike languages like C and C++, a whole category of *memory safety* bugs do *not* occur, because you're not manually messing with memory. But you may still experience other language nightmares, like *thread safety* (see Chapter 23).

The following are debugging techniques, some quick and others more involved.

Assert

Use `assert` to declare that something must be true and to otherwise raise an exception on the spot. Assertions are often useful early in your coding progress, as you're feeling your way through and are lacking checks and tests that will come later. Often you use one to check for "impossible" data that would cause an exception, like a division by zero, or a `None` where a valid value is expected.

You assert that something is true. If it isn't, an `AssertionError` is raised, with some details of what failed:

```
>>> x = 5
>>> assert x == 5
>>> assert x > 4
>>> assert x == 0
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    assert x == 0
           ^^^^^^
AssertionError
```

You can include a string after the test expression, and it will also be printed if the assertion fails:

```
>>> x = 5
>>> assert x == 0, "I thought x was zero!"
Traceback (most recent call last):
  File "<python-input-5>", line 1, in <module>
    assert x == 0, "I thought x was zero!"
           ^^^^^^
AssertionError: I thought x was zero!
```

Assertion checking can be disabled by passing the `-O` option to the Python interpreter.

Print

The simplest way to debug in Python is to print out strings. Some useful things to print include `vars()`, which extracts the values of your local variables, including function arguments:

```
>>> def func(*args, **kwargs):
...     print(vars())
...
>>> func(1, 2, 3)
{'args': (1, 2, 3), 'kwargs': {}}
>>> func(['a', 'b', 'argh'])
{'args': (['a', 'b', 'argh'],), 'kwargs': {}}
```

Other variables that are often worth printing are returned by `locals()` and `globals()`.

If your code also includes normal prints to standard output, you can write your debugging messages to standard error output with `print(`*`stuff`*`, file=sys.stderr)`.

F-strings

"Newest Style: f-strings" on page 69 mentioned that Python 3.6 introduced the wonderful *f-string*. Any expression that you enclose with curly braces (`{}`) will be evaluated and interpolated into the string:

```
>>> x = "thing"
>>> print(f"x = {x}")
x = thing
```

F-strings also have a particularly nice debug print feature. If you put an = sign somewhere before the final } curly brace that ends the f-string expression (you can have spaces there), Python will print the whole expression *before* the =, then the value that it expresses:

```
>>> x = "thing"
>>> print(f"{x=}")
x='thing'
>>> print(f"{  x =  }")
  x =  'thing'
>>> print(f"{ 10/2 = }")
10/2 = 5.0
```

Pprint()

All our examples have used `print()` (or just the variable name, in the interactive interpreter) to print results. Sometimes, they are hard to read. We need a *pretty printer* such as `pprint()`, which is in the standard library:

```
>>> from pprint import pprint
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
```

Plain old `print()` just dumps whatever you include out there:

```
>>> print(quotes)
OrderedDict([('Moe', 'A wise guy, huh?'), ('Larry', 'Ow!'),
    ('Curly', 'Nyuk nyuk!')])
```

However, `pprint()` tries to align elements for better readability:

```
>>> pprint(quotes)
{'Moe': 'A wise guy, huh?',
 'Larry': 'Ow!',
 'Curly': 'Nyuk nyuk!'}
```

IceCream

Install the useful third-party IceCream package with `pip install icecream`. It has aspects of other tools in this chapter:

F-strings
: Print the code expression text as well as the value.

Logging
: Print the current module and line.

Decorators
: Print the current function name, arguments, and returns.

Pretty-print
: Use `pprint()` to dump readable values.

I'll import `ic()` from `icecream`. Calling `ic()` with no arguments prints the filename, function, and line:

```
>>> from icecream import ic
>>> def sub1():
...     ic()
...     print("in sub1")
```

```
...
>>> def sub2():
...     ic()
...     print("in sub2")
...
>>> ic()
ic| <python-input-5>:1 in <module> at 14:58:43.205
>>> sub1()
ic| <python-input-1>:2 in sub1() at 14:58:51.074
in sub1
>>> sub2()
ic| <python-input-2>:2 in sub2() at 14:58:56.130
in sub2
```

The `ic()` function evaluates and prints any arguments you pass it, as well as any return value:

```
>>> ic(int)
ic| int: <class 'int'>
<class 'int'>
>>> ic(int("5"))
ic| int("5"): 5
5
>>> ic(list(range(5)))
ic| list(range(5)): [0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

The `ic()` function also returns its arguments, so you can stick it in the middle of almost any expression.

Decorators

As you read in "Decorators" on page 170, a decorator can call code before or after a function without modifying the code within the function itself. This means that you can use a decorator to do something before or after any Python function, not just ones that you wrote. Let's define the decorator `dump` to print the input arguments and output values of any function as it's called (designers know that a dump often needs decorating), as shown in Example 16-1.

Example 16-1. dump.py

```
def dump(func):
    "Print input arguments and output value(s)"
    def wrapped(*args, **kwargs):
        print("Function name:", func.__name__)
        print("Input arguments:", ' '.join(map(str, args)))
        print("Input keyword arguments:", kwargs.items())
        output = func(*args, **kwargs)
        print("Output:", output)
        return output
    return wrapped
```

Now the decorated function: `double()` expects numeric arguments, either named or unnamed, and returns them in a list with their values doubled (Example 16-2).

Example 16-2. test_dump.py

```
from dump import dump

@dump
def double(*args, **kwargs):
    "Double every argument"
    output_list = [ 2 * arg for arg in args ]
    output_dict =  { k:2*v for k,v in kwargs.items() }
    return output_list, output_dict

if __name__ == '__main__':
    output = double(3, 5, first=100, next=98.6, last=-40)
```

Take a moment to run this test:

```
$ python test_dump.py
Function name: double
Input arguments: 3 5
Input keyword arguments: dict_items([(first, 100), (next, 98.6),
    (last, -40)])
Output: ([6, 10], {first: 200, next: 197.2, last: -80})
```

Logging

At some point you might need to graduate from using `print()` statements to logging messages. A *log* is usually a system file that accumulates messages, often inserting useful information such as a timestamp or the name of the user who's running the program. Often logs are *rotated* (renamed) daily and compressed; by doing so, they don't fill up your disk and cause problems themselves. When something goes wrong with your program, you can look at the appropriate logfile to see what happened. The contents of exceptions are especially useful in logs because they show you the actual line at which your program croaked, and why.

The standard Python library module is `logging`. I've found most descriptions of it somewhat confusing. After a while it makes more sense, but it does seem overly complicated at first. The `logging` module includes these concepts:

- The *message* that you want to save to the log
- Ranked priority *levels* and matching functions: `debug()`, `info()`, `warn()`, `error()`, and `critical()`
- One or more *logger* objects as the main connection with the module

- *Handlers* that direct the message to your terminal, a file, a database, or somewhere else
- *Formatters* that create the output
- *Filters* that make decisions based on the input

For the simplest logging example, just import the module and use some of its functions:

```
>>> import logging
>>> logging.debug("Looks like rain")
>>> logging.info("And hail")
>>> logging.warn("Did I hear thunder?")
WARNING:root:Did I hear thunder?
>>> logging.error("Was that lightning?")
ERROR:root:Was that lightning?
>>> logging.critical("Stop fencing and get inside!")
CRITICAL:root:Stop fencing and get inside!
```

Did you notice that `debug()` and `info()` didn't do anything, and the other two printed *LEVEL*:`root`: before each message? So far, it's like a `print()` statement with multiple personalities, some of them hostile.

But it is useful. You can scan for a particular value of *LEVEL* in a logfile to find certain messages, compare timestamps to see what happened before your server crashed, and so on.

A lot of digging through the documentation answers the first mystery (we get to the second one in a page or two): the default priority *level* is `WARNING`, and that got locked in as soon as we called the first function (`logging.debug()`). We can set the default level by using `basicConfig()`. `DEBUG` is the lowest level, so this enables it and all the higher levels to flow through:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> logging.debug("It's raining again")
DEBUG:root:It's raining again
>>> logging.info("With hail the size of hailstones")
INFO:root:With hail the size of hailstones
```

We did all that with the default `logging` functions without actually creating a *logger* object. Each logger has a name. Let's make one called `bunyan`:

```
>>> import logging
>>> logging.basicConfig(level='DEBUG')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug('Timber!')
DEBUG:bunyan:Timber!
```

If the logger name contains any dot characters, they separate levels of a hierarchy of loggers, each with potentially different properties. This means that a logger named

`quark` is higher than one named `quark.charmed`. The special *root logger* is at the top and is called `''`.

So far, we've just printed messages, which is not a great improvement over `print()`. We use *handlers* to direct the messages to different places. The most common is a *logfile*, and here's how you do it:

```
>>> import logging
>>> logging.basicConfig(level='DEBUG', filename='blue_ox.log')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug("Where's my axe?")
>>> logger.warn("I need my axe")
>>>
```

Aha! The lines aren't on the screen anymore; instead, they're in the file named *blue_ox.log*:

```
DEBUG:bunyan:Where's my axe?
WARNING:bunyan:I need my axe
```

Calling `basicConfig()` with a `filename` argument created a `FileHandler` for you and made it available to your logger. The `logging` module includes at least 15 handlers to send messages to places such as email and web servers as well as the screen and files.

Finally, you can control the *format* of your logged messages. In our first example, our default gave us something similar to this:

```
WARNING:root:Message...
```

If you provide a `format` string to `basicConfig()`, you can change to the format of your preference:

```
>>> import logging
>>> fmt = '%(asctime)s %(levelname)s %(lineno)s %(message)s'
>>> logging.basicConfig(level='DEBUG', format=fmt)
>>> logger = logging.getLogger('bunyan')
>>> logger.error("Where's my other plaid shirt?")
2025-07-31 16:14:15,521 ERROR 1 Where's my other plaid shirt?
```

We let the logger send output to the screen again but change the format.

The `logging` module recognizes a number of variable names in the `fmt` format string. We use `asctime` (date and time as an ISO 8601 string), `levelname`, `lineno` (line number), and the `message` itself. There are other built-ins, and you can provide your own variables as well.

There's much more to `logging` than this little overview can provide. You can log to more than one place at the same time, with different priorities and formats. The package has a lot of flexibility but sometimes at the cost of simplicity.

Pdb

All the techniques mentioned so far help, but sometimes there's no substitute for a real debugger. Most IDEs include a debugger, with varying features and user interfaces. Here, I describe uses of the standard Python debugger, pdb.

If you run your program with the `-i` flag, Python will drop you into its interactive interpreter if the program fails.

Here's a program with a bug that depends on data—the kind of bug that can be particularly hard to find. This is a real bug from the early days of computing, and it baffled programmers for quite a while.

We're going to read a file of countries and their capital cities, separated by a comma, and write them out as *capital*, *country*. They might be capitalized incorrectly, so we should fix that also when we print. Oh, and there might be extra spaces here and there, and you'll want to get rid of those too. Finally, although it would make sense for the program to just read to the end of the file, for some reason our manager told us to stop when we encounter the word `quit` (in any mixture of uppercase and lowercase characters). Example 16-3 shows a sample data file.

Example 16-3. cities.csv

```
France, Paris
venuzuela,caracas
  LithuaniA,vilnius
     quit
```

Let's design our *algorithm* (method for solving the problem). This is *pseudocode*: it looks like a program, but is just a way to explain the logic in normal language before converting it to an actual program. One reason programmers like Python is because it *looks a lot like pseudocode*, so there's less work involved when it's time to convert it to a working program:

```
    for each line in the text file:
        read the line
        strip leading and trailing spaces
        if `quit` occurs in the lower-case copy of the line:
            stop
        else:
            split the country and capital by the comma character
            trim any leading and trailing spaces
            convert the country and capital to titlecase
            print the capital, a comma, and the country
```

We need to strip initial and trailing spaces from the names because that was a requirement. Likewise for the lowercase comparison with `quit` and converting the city and country names to title case. That being the case, let's whip out *capitals.py*, which is sure to work perfectly (Example 16-4).

Example 16-4. capitals.py

```
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
            line = line.strip()
            if 'quit' in line.lower():
                return
            country, city = line.split(',')
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')

if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

Let's test *capitals.py* with that sample data file we made earlier. Ready, fire, aim:

```
$ python capitals.py cities.csv
Paris,France
Caracas,Venezuela
Vilnius,Lithuania
```

Looks great! The *capitals.py* program passed one test, so let's put it in production, processing capitals and countries from around the world—until it fails, but only for this data file (Example 16-5).

Example 16-5. cities2.csv

```
argentina,buenos aires
bolivia,la paz
brazil,brasilia
chile,santiago
colombia,Bogotá
ecuador,quito
falkland islands,stanley
french guiana,cayenne
guyana,georgetown
paraguay,Asunción
peru,lima
suriname,paramaribo
uruguay,montevideo
venezuela,caracas
quit
```

The program ends after printing only 5 lines of the 15 in the data file, as demonstrated here:

```
$ python capitals.py cities2.csv
Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
```

What happened? We can keep editing *capitals.py*, putting `print()` statements in likely places, but let's see if the debugger can help us.

To use the debugger, import the `pdb` module from the command line by typing **`-m pdb`**, like so:

```
$ python -m pdb capitals.py cities2.csv
> /.../capitals.py(1)<module>()
-> def process_cities(filename):
(Pdb)
```

This starts the program and places you at the first line. If you type **`c`** (*continue*), the program will run until it ends, either normally or with an error:

```
(Pdb) c
Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
The program finished and will be restarted
> /.../capitals.py(1)<module>()
-> def process_cities(filename):
```

The program completes normally, just as it did when we ran it earlier outside of the debugger. Let's try again, using commands to narrow down where the problem lies. It seems to be a *logic error* rather than a syntax problem or exception (which would have printed error messages).

Type **`s`** (*step*) to single-step through Python lines. This steps through *all* Python code lines: yours, the standard library's, and any other modules you might be using. When you use `s`, you also go into functions and single-step within them. Type **`n`** (*next*) to single-step but *not* go inside functions; when you get to a function, a single `n` causes the entire function to execute and take you to the next line of your program. Thus, use `s` when you're not sure where the problem is; use `n` when you're sure that a particular function isn't the cause, especially if it's a long function. Often you'll single-step through your own code and step over library code, which is presumably well tested.

Let's use `s` to step from the beginning of the program into the function `process_cities()`. (I'm using an ellipsis (…) instead of the actual directory that the example code used on my computer):

```
(Pdb) s
> /.../capitals.py(12)<module>()
 -> if __name__ == '__main__':
(Pdb) s
 > /.../capitals.py(13)<module>()
 -> import sys
(Pdb) s
 > /.../capitals.py(14)<module>()
 -> process_cities(sys.argv[1])
(Pdb) s
 --Call--
 > /.../capitals.py(1)process_cities()
 -> def process_cities(filename):
(Pdb) s
 > /.../capitals.py(2)process_cities()
 -> with open(filename, 'rt') as file:
```

Type **`l`** (*list*) to see the next few lines of your program:

```
(Pdb) l
  1     def process_cities(filename):
  2  ->     with open(filename, 'rt') as file:
  3             for line in file:
  4                 line = line.strip()
  5                 if 'quit' in line.lower():
  6                     return
  7                 country, city = line.split(',')
  8                 city = city.strip()
  9                 country = country.strip()
 10                 print(city.title(), country.title(), sep=',')
 11
(Pdb)
```

The arrow (`->`) denotes the current line.

We could continue using `s` or `n`, hoping to spot something, but let's use one of the main features of a debugger: breakpoints. A *breakpoint* stops execution at the line you indicate. In our case, we want to know why `process_cities()` bails out before it has read all the input lines. Line 3 (`for line in file:`) will read every line in the input file, so that seems innocent. The only other place where we could return from the function before reading all the data is at line 6 (`return`).

Let's set a breakpoint on line 6:

```
(Pdb) b 6
Breakpoint 1 at /.../capitals.py:6
```

Next, let's continue the program until it either hits the breakpoint or reads all the input lines and finishes normally:

```
(Pdb) c
 Buenos Aires,Argentina
 La Paz,Bolivia
 Brasilia,Brazil
 Santiago,Chile
 Bogotá,Colombia
 > /.../capitals.py(6)process_cities()
 -> return
```

Aha! The program stops at our line 6 breakpoint. This indicates that the program wants to return early after reading the country after Colombia. Let's print the value of `line` to see what we just read:

```
(Pdb) p line
ecuador,quito
```

What's so special about—oh, never mind.

Really? **`quit`**o? Our manager never expected the string `quit` to turn up inside normal data, so using it as a *sentinel* (end indicator) value like this was a boneheaded idea. You march right in there and tell him that—while I wait here.

If at this point you still have a job, you can see all your breakpoints by using a plain `b` command:

```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at
   /.../capitals.py:6
    breakpoint already hit 1 time
```

An `l` will show your code lines, the current line (`->`), and any breakpoints (`B`). A plain `l` will start listing from the end of your previous call to `l`, so include the optional starting line (here, let's start from line `1`):

```
(Pdb) l 1
  1     def process_cities(filename):
  2         with open(filename, rt) as file:
  3             for line in file:
  4                 line = line.strip()
  5                 if quit in line.lower():
  6 B->                 return
  7                 country, city = line.split(,)
  8                 city = city.strip()
  9                 country = country.strip()
 10                 print(city.title(), country.title(), sep=,)
 11
```

OK, as shown in Example 16-6, let's fix that `quit` test to match only the full line, not within other characters.

Example 16-6. capitals2.py

```
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
            line = line.strip()
            if 'quit' == line.lower():
                return
            country, city = line.split(',')
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')

if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

Once more, with feeling:

```
$ python capitals2.py cities2.csv
Buenos Aires,Argentina
La Paz,Bolivia
Brasilia,Brazil
Santiago,Chile
Bogotá,Colombia
Quito,Ecuador
Stanley,Falkland Islands
Cayenne,French Guiana
Georgetown,Guyana
Asunción,Paraguay
Lima,Peru
Paramaribo,Suriname
Montevideo,Uruguay
Caracas,Venezuela
```

That was a skimpy overview of the debugger, just enough to show you what it can do and the commands you'd use most of the time. Remember: more tests, less debugging.

Breakpoint()

Python 3.7 has a new built-in function called `breakpoint()`. If you add it to your code, a debugger will automatically start up and pause at each location. Without this, you would need to fire up a debugger like `pdb` and set breakpoints manually, as you saw earlier.

The default debugger is the one you've just seen (`pdb`), but this can be changed by setting the environment variable `PYTHONBREAKPOINT`. For example, you could specify use of the web-based remote debugger Web-PDB:

```
$ export PYTHONBREAKPOINT=web_pdb.set_trace
```

The official documentation is a bit dry, but there are good overviews in "Python 3.7's Built-in Breakpoint—A Quick Tour" by Anthony Shaw and "Python breakpoint()" by Pankaj Kumar.

Review/Preview

You (or someone you trusted) broke it, you fix it. Debugging can be frustrating, but it can also be rewarding to diagnose and fix a problem.

The next chapter introduces Part III: real-life projects you can build with the tools you learned in Part II.

Practice

16.1 Use the f-string debug print feature to print these expressions:

- Assign 5 to `x`, then print double it if `x` is odd; otherwise, print `2`.
- Print `x` using 20 spaces of width, center adjusted, with dots filling the left and right spaces.

PART III

Quests

Armed with our knightly training and trusty tools, we now issue forth from the castle on a variety of noble quests. Surely our fates will be glorious!

CHAPTER 17
Text Data

I live for the text. It's my job.
—Ian McKellen

You saw the basics of Python strings in Chapter 4. Now it's time to dig deeper into strings and text data.

Text Strings: Unicode

Python 3 strings are Unicode character sequences, not bytearrays. This is, by far, the single largest language change from Python 2.

All the text examples in this book thus far have been plain old ASCII. ASCII was defined in the 1960s, before mullets roamed the earth. Computers then were the size of refrigerators, and only slightly smarter.

As I mentioned way back in Chapter 2, the basic unit of computer storage is the *byte*, which can store 256 unique values in its eight *bits*. For various reasons, ASCII used only seven bits (128 unique values): 26 uppercase letters, 26 lowercase letters, 10 digits, some punctuation symbols, some spacing characters, and some nonprinting control codes.

Unfortunately, the world has more letters than ASCII provides. You could have a hot dog at a diner, but never a Gewürztraminer at a café.[1] Many attempts have been made to cram more letters and symbols into eight bits, and you'll see them at times.

1 Gewürztraminer wine has an umlaut in Germany, but loses it in Alsace on the way to France.

Here are just a couple of those:

- *Latin-1*, or *ISO 8859-1*
- Windows code page *1252*

Each of these uses all eight bits, but even that's not enough, especially when you need non-European languages. *Unicode* is an ongoing international standard to define the characters of all the world's languages, plus symbols from mathematics and other fields. And emojis!

> Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.
>
> —The Unicode Consortium

The Unicode Code Charts page has links to all the currently defined character sets with images. The latest version (16.0) defines more than 155,000 characters, each with a unique name and identification number. Python handles all of these. Character places are divided into 17 *planes*, each containing 65,536 *code points*. The first plane, the *Basic Multilingual Plane*, has most of the characters of the world's languages. See the Wikipedia pages about Unicode planes and characters for details.

Python Unicode Strings

If you know the Unicode ID or name for a character, you can use it in a Python string. Here are some examples:

- A `\u` followed by *four* hex numbers specifies a character in one of Unicode's 256 basic multilingual planes.[2] The first two are the plane number (`00` to `FF`), and the next two are the index of the character within the plane. Plane `00` is good old ASCII, and the character positions within that plane are the same as ASCII.
- For characters in the higher planes, we need more bits. The Python escape sequence for these is `\U` followed by *eight* hex characters; the leftmost ones need to be `0`.
- For all characters, `\N{`*name*`}` lets you specify it by its standard *name*. The Unicode Character Name Index page lists these.

2 Hex numbers use base 16, specified with characters `0-9` and `A-F`.

The Python `unicodedata` module has functions that translate in both directions:

`lookup()`
: Takes a case-insensitive name and returns a Unicode character

`name()`
: Takes a Unicode character and returns an uppercase name

In the following example, we'll write a test function that takes a Python Unicode character, looks up its name, and looks up the character again from the name. This character should match the original. Note that the `print()` statement uses a cool debugging property of f-strings; see "F-strings" on page 273:

```
>>> def unicode_test(value):
...     import unicodedata
...     name = unicodedata.name(value)
...     value2 = unicodedata.lookup(name)
...     print(f'{value=}, {name=}, {value2=}')
...
```

Let's try some characters, beginning with a plain ASCII letter:

```
>>> unicode_test('A')
value="A", name="LATIN CAPITAL LETTER A", value2="A"
```

ASCII punctuation:

```
>>> unicode_test('$')
value="$", name="DOLLAR SIGN", value2="$"
```

A Unicode currency character:

```
>>> unicode_test('\u00a2')
value="¢", name="CENT SIGN", value2="¢"
```

Another Unicode currency character:

```
>>> unicode_test('\u20ac')
value="€", name="EURO SIGN", value2="€"
```

The only problem you could potentially run into is limitations in the font you're using to display text. Few fonts have images for all Unicode characters, and might display a placeholder character for missing ones. For instance, here's the Unicode symbol for SNOWMAN, like symbols in dingbat fonts:

```
>>> unicode_test('\u2603')
value="☃", name="SNOWMAN", value2="☃"
```

Suppose that we want to save the word `café` in a Python string. One way is to copy and paste it from a file or website and hope that it works:

```
>>> place = 'café'
>>> place
'café'
```

This works because I copied and pasted from a source that used UTF-8 encoding (which we'll look at in a few pages) for its text.

How can we specify that final é character? If you look at the character index for E, you see that the name `E WITH ACUTE, LATIN SMALL LETTER` has the hex value `00E9`.

Next, use that name from the Unicode site to look up the code:

```
>>> unicodedata.lookup('E WITH ACUTE, LATIN SMALL LETTER')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "undefined character name 'E WITH ACUTE, LATIN SMALL LETTER'"
```

What happened?

The names listed on the Unicode Character Name Index page were reformatted to make them sort nicely for display. To convert them to their real Unicode names (the ones that Python uses), remove the comma and move the part of the name that was after the comma to the beginning. Accordingly, change `E WITH ACUTE, LATIN SMALL LETTER` to `LATIN SMALL LETTER E WITH ACUTE`:

```
>>> unicodedata.lookup('LATIN SMALL LETTER E WITH ACUTE')
'é'
```

The string name that you provide to `lookup()` (or inside the `\N{...}` in a string) is case-independent, so `latin small letter e with acute` would work too.

To get the hex Unicode value from a name:

```
>>> hex(ord('\N{snowman}'))
'0x2603'
```

If you know the hex Unicode value, use Python's Unicode `name()` function to go the other way:

```
>>> unicodedata.name('\u2603')
'SNOWMAN'
```

Now we can specify the string `café` by code or by name:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> place = 'caf\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> place
'café'
```

In the preceding snippet, we inserted the é directly in the string, but we can also build a string by appending:

```
>>> u_umlaut = '\N{LATIN SMALL LETTER U WITH DIAERESIS}'
>>> u_umlaut
'ü'
>>> drink = 'Gew' + u_umlaut + 'rztraminer'
```

```
>>> print('Now I can finally have my', drink, 'in a', place)
Now I can finally have my Gewürztraminer in a café
```

The string `len()` function counts Unicode *code points*, not bytes:

```
>>> len('$')
1
>>> len('\U0001f47b')
1
```

If you know the Unicode numeric ID, you can use the standard `chr()` function to quickly convert between integer IDs and single-character Unicode strings:

```
>>> chr(233)
'é'
>>> chr(0xe9)
'é'
>>> chr(0x1fc6)
'ῆ'
```

UTF-8

You don't need to worry about how Python stores each Unicode character when you do normal string processing. However, when you exchange data with the outside world, you need a couple of things:

- A way to *encode* character strings to bytes
- A way to *decode* bytes to character strings

If there were fewer than 65,536 characters in Unicode, we could stuff each Unicode character ID into two bytes. Unfortunately, there are more. We could encode every ID into four bytes, but that would increase the memory and disk storage space needs for common text strings by four times.

Ken Thompson and Rob Pike, whose names will be familiar to Unix developers, designed the *UTF-8* dynamic encoding scheme one night on a placemat in a New Jersey diner. It uses one to four bytes per Unicode character:

- One byte for ASCII
- Two bytes for most Latin-derived (but not Cyrillic) languages
- Three bytes for the rest of the basic multilingual plane
- Four bytes for the rest, including some Asian languages and symbols

UTF-8 is the standard text encoding in Python, Linux, and HTML. It's fast, complete, and works well. If you use UTF-8 encoding throughout your code, life will be much easier than trying to hop in and out of various encodings.

If you create a Python string by copying and pasting from another source such as a web page, be sure the source is encoded in the UTF-8 format. Sometimes text that was encoded as Latin-1 or Windows 1252 is copied into a Python string, which causes an exception later with an invalid byte sequence.

Encode

You *encode* a *string* to *bytes*. The string `encode()` function's first argument is the encoding name. The choices include those presented in Table 17-1.

Table 17-1. Encodings

Encoding name	Description
`ascii`	Good old seven-bit ASCII
`utf-8`	Eight-bit variable-length encoding, and what you almost always want to use
`latin-1`	Also known as ISO 8859-1
`cp-1252`	A common Windows encoding
`unicode-escape`	Python Unicode literal format, `\u__xxxx__` or `\U__xxxxxxxx__`

You can encode anything as UTF-8. Let's assign the Unicode string `\u2603` to the name `snowman`:

```
>>> snowman = '\u2603'
```

`snowman` is a Python Unicode string with a single character, regardless of the number of bytes that might be needed to store it internally:

```
>>> len(snowman)
1
```

Next, let's encode this Unicode character to a sequence of bytes:

```
>>> ds = snowman.encode('utf-8')
```

As I mentioned earlier, UTF-8 is a variable-length encoding. In this case, it uses three bytes to encode the single `snowman` Unicode character:

```
>>> len(ds)
3
>>> ds
b'\xe2\x98\x83'
```

Now, `len()` returns the number of bytes (3) because `ds` is a `bytes` variable.

You can use encodings other than UTF-8, but you'll get errors if the Unicode string can't be handled by the encoding. For example, if you use the `ascii` encoding, it will fail unless your Unicode characters happen to be valid ASCII characters as well:

```
>>> ds = snowman.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u2603'
in position 0: ordinal not in range(128)
```

The `encode()` function takes a second argument to help you avoid encoding exceptions. Its default value, which you can see in the previous example, is `strict`; it raises a `UnicodeEncodeError` if it sees a non-ASCII character. There are other encodings. Use `ignore` to throw away anything that won't encode:

```
>>> snowman.encode('ascii', 'ignore')
b''
```

Use `replace` to substitute `?` for unknown characters:

```
>>> snowman.encode('ascii', 'replace')
b'?'
```

Use `backslashreplace` to produce a Python Unicode character string, like `unicode-escape`:

```
>>> snowman.encode('ascii', 'backslashreplace')
b'\\u2603'
```

You would use this if you needed a printable version of the Unicode escape sequence.

Use `xmlcharrefreplace` to make HTML-safe strings:

```
>>> snowman.encode('ascii', 'xmlcharrefreplace')
b'&#9731;'
```

I'll provide more details on HTML conversion in "HTML Entities" on page 297.

Decode

We *decode* byte strings to Unicode text strings. Whenever we get text from an external source (files, databases, websites, network APIs, and so on), it's encoded as byte strings. The tricky part is knowing which encoding was used, so we can *run it backward* and get Unicode strings.

The problem is that nothing in the byte string itself says which encoding was used. I mentioned the perils of copying and pasting from websites earlier. You've probably visited websites with odd characters where plain old ASCII characters should be.

Let's create a Unicode string called `place` with the value `café`:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> type(place)
<class 'str'>
```

Encode it in UTF-8 format in a `bytes` variable called `place_bytes`:

```
>>> place_bytes = place.encode('utf-8')
>>> place_bytes
b'caf\xc3\xa9'
>>> type(place_bytes)
<class 'bytes'>
```

Notice that `place_bytes` has five bytes. The first three are the same as ASCII (a strength of UTF-8), and the final two encode the `é`. Now let's decode that byte string back to a Unicode string:

```
>>> place2 = place_bytes.decode('utf-8')
>>> place2
'café'
```

This works because we encoded to UTF-8 and decoded from UTF-8. What if we told it to decode from another encoding?

```
>>> place3 = place_bytes.decode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3:
ordinal not in range(128)
```

The ASCII decoder throws an exception because the byte value `0xc3` is illegal in ASCII. There are some 8-bit character set encodings in which values between 128 (hex `80`) and 255 (hex `FF`) are legal but not the same as UTF-8:

```
>>> place4 = place_bytes.decode('latin-1')
>>> place4
'cafÃ©'
>>> place5 = place_bytes.decode('windows-1252')
>>> place5
'cafÃ©'
```

Urk.

The moral of this story: whenever possible, use UTF-8 encoding. It works, is supported everywhere, can express every Unicode character, and is quickly decoded and encoded.

Even though you can specify any Unicode character, that doesn't mean that your computer will display all of them. That depends on the *font* that you're using, which may display nothing or a fill-in image for many characters. Apple created the Last Resort Font for the Unicode Consortium, and uses it in its own operating systems. The "Fallback Font" page on Wikipedia has a few more details. Another font with everything between `\u0000` and `\uffff`, and a few more, is Unifont.

HTML Entities

Python 3.4 added another way to convert to and from Unicode but using HTML *character entities*.[3] This may be easier to use than looking up Unicode names, especially if you're working on the web:

```
>>> import html
>>> html.unescape("&egrave;")
'è'
```

This conversion also works with numbered entities, decimal or hex:

```
>>> import html
>>> html.unescape("&#233;")
'é'
>>> html.unescape("&#xe9;")
'é'
```

You can even import the named entity translations as a dictionary and do the conversion yourself. Drop the initial `&` for the dictionary key (you can also drop the final `;`, but it seems to work either way):

```
>>> from html.entities import html5
>>> html5["egrave"]
'è'
>>> html5["egrave;"]
'è'
```

To go the other direction (from a single Python Unicode character to an HTML entity name), first get the decimal value of the character with `ord()`:

```
>>> import html
>>> char = '\u00e9'
>>> dec_value = ord(char)
>>> html.entities.codepoint2name[dec_value]
'eacute'
```

3 See the HTML5 named-character reference chart.

For Unicode strings with more than one character, use this two-step conversion:

```
>>> place = 'caf\u00e9'
>>> byte_value = place.encode('ascii', 'xmlcharrefreplace')
>>> byte_value
b'caf&#233;'
>>> byte_value.decode()
'caf&#233;'
```

The expression `place.encode('ascii', 'xmlcharrefreplace')` returns ASCII characters but as type `bytes` (because it *en*coded). The following `byte_value.decode()` is needed to convert `byte_value` to an HTML-compatible string.

Normalization

Some Unicode characters can be represented by more than one Unicode encoding. They'll look the same but won't compare the same because they have different internal byte sequences. For example, take the acute accented `é` in `café`. Let's make a single-character `é` in multiple ways:

```
>>> eacute1 = 'é'                              # UTF-8, pasted
>>> eacute2 = '\u00e9'                         # Unicode code point
>>> eacute3 = \                                # Unicode name
...     '\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> eacute4 = chr(233)                         # decimal byte value
>>> eacute5 = chr(0xe9)                        # hex byte value
>>> eacute1, eacute2, eacute3, eacute4, eacute5
('é', 'é', 'é', 'é', 'é')
>>> eacute1 == eacute2 == eacute3 == eacute4 == eacute5
True
```

Try a few sanity checks:

```
>>> import unicodedata
>>> unicodedata.name(eacute1)
'LATIN SMALL LETTER E WITH ACUTE'
>>> ord(eacute1)            # as a decimal integer
233
>>> 0xe9                    # Unicode hex integer
233
```

Now let's make an accented `e` by combining a plain `e` with an acute accent:

```
>>> eacute_combined1 = "e\u0301"
>>> eacute_combined2 = "e\N{COMBINING ACUTE ACCENT}"
>>> eacute_combined3 = "e" + "\u0301"
>>> eacute_combined1, eacute_combined2, eacute_combined3
('é', 'é', 'é'))
>>> eacute_combined1 == eacute_combined2 == eacute_combined3
True
>>> len(eacute_combined1)
2
```

We built a Unicode character from two characters, and it looks the same as the original é. But as they say on Sesame Street, one of these things is not like the other:

```
>>> eacute1 == eacute_combined1
False
```

If you had two Unicode text strings from different sources, one using `eacute1` and another `eacute_combined1`, they would appear the same but would mysteriously not act the same.

You can fix this with the `normalize()` function in the `unicodedata` module:

```
>>> import unicodedata
>>> eacute_normalized = unicodedata.normalize('NFC', eacute_combined1)
>>> len(eacute_normalized)
1
>>> eacute_normalized == eacute1
True
>>> unicodedata.name(eacute_normalized)
'LATIN SMALL LETTER E WITH ACUTE'
```

That `NFC` means *normal form, composed*.

If you would like to learn more about Unicode, these links are particularly helpful:

- Unicode HOWTO
- "Pragmatic Unicode" by Ned Batchelder
- "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" by Joel Spolsky

Text Strings: Regular Expressions

Chapter 4 discussed simple string operations. Armed with that introductory information, you've probably used simple "wildcard" patterns on the command line, such as the Unix command `ls *.py`, which means *list all filenames ending in .py*.

It's time to explore more complex pattern matching by using *regular expressions*. These are provided in the standard module `re`, which we'll import. You define a string *pattern* that you want to match, and the *source* string to match against. For simple matches, usage looks like this:

```
>>> import re
>>> result = re.match('You', 'Young Frankenstein')
```

Here, `You` is the *pattern* we're looking for, and `Young Frankenstein` is the *source* (the string we want to search). The `match()` function checks whether the *source* begins with the *pattern*.

For more complex matches, you can *compile* your pattern first to speed up the match later:

```
>>> import re
>>> youpattern = re.compile('You')
```

Then you can perform your match against the compiled pattern:

```
>>> result = youpattern.match('Young Frankenstein')
```

Because this is a common Python gotcha, I'll say it again here: `match()` matches only a pattern starting at the *beginning* of the source. The `search()` function matches a pattern *anywhere* in the source.

The `match()` function is not the only way to compare the pattern and source. Here are several other methods you can use, and I'll discuss each in the following sections:

- `search()` returns the first match, if any.
- `findall()` returns a list of all nonoverlapping matches, if any.
- `split()` splits the *source* at matches with *pattern* and returns a list of the string pieces.
- `sub()` takes another *replacement* argument, and changes all parts of the *source* that are matched by *pattern* to *replacement*.

Most of the regular expression examples here use ASCII, but Python's string functions, including regular expressions, work with any Python string and any Unicode characters.

Find Exact Beginning Match with match()

Does the string `Young Frankenstein` begin with the word `You`? Here's some code with comments:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('You', source)  # match starts at the beginning of source
>>> if m:  # match returns an object; do this to see what matched
...     print(m.group())
...
```

```
You
>>> m = re.match('^You', source) # start anchor does the same
>>> if m:
...     print(m.group())
...
You
```

How about `Frank`?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('Frank', source)
>>> if m:
...     print(m.group())
...
```

This time, `match()` returned nothing, so the `if` did not run the `print` statement.

As I mentioned in "New: I Am the Walrus" on page 88, as of Python 3.8, you can shorten this example with the walrus operator:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> if m := re.match('Frank', source):
...     print(m.group())
...
```

OK, now let's use `search()` to see whether `Frank` is anywhere in the source string:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.search('Frank', source)
>>> if m:
...      print(m.group())
...
Frank
```

Let's change the pattern and try a beginning match with `match()` again:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('.*Frank', source)
>>> if m:  # match returns an object
...     print(m.group())
...
Young Frank
```

Here's a brief explanation of how our new `.*Frank` pattern works:

- `.` means *any single character*.
- `*` means *zero or more of the preceding thing*. Together, `.*` means *any number of characters* (even zero).
- `Frank` is the phrase that we wanted to match, somewhere.

`match()` returns the string that matches `.*Frank`, which is `Young Frank`.

Find First Match with search()

You can use `search()` to find the pattern `Frank` anywhere in the source string `Young Frankenstein`, without the need for the `.*` wildcards:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.search('Frank', source)
>>> if m:  # search returns an object
...     print(m.group())
...
Frank
```

Find All Matches with findall()

The preceding examples looked for one match only. But what if you want to know how many instances of the single-letter string `n` are in the string?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n', source)
>>> m   # findall returns a list
['n', 'n', 'n', 'n']
>>> print('Found', len(m), 'matches')
Found 4 matches
```

How about `n` followed by any character?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n.', source)
>>> m
['ng', 'nk', 'ns']
```

Notice that it did not match that final `n`. We need to say that the character after `n` is optional, with `?`:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n.?', source)
>>> m
['ng', 'nk', 'ns', 'n']
```

Split at Matches with split()

The next example shows you how to split a string into a list by a pattern rather than a simple string (as the normal string `split()` method would do):

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.split('n', source)
```

```
>>> m    # split returns a list
['You', 'g Fra', 'ke', 'stei', '']
```

Replace at Matches with sub()

This is like the string `replace()` method, but for patterns rather than literal strings:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.sub('n', '?', source)
>>> m   # sub returns a string
'You?g Fra?ke?stei?'
```

Patterns

Regular expressions are a little language of their own, comprised of many awkward-looking character patterns. Let's look at the most useful ones.

Using Special Characters

Many descriptions of regular expressions start with all the details of how to define them. I think that's a mistake. Regular expressions are a not-so-little language in their own right, with too many details to fit in your head at once. They use so much punctuation that they look like cartoon characters swearing.

With these expressions (`match()`, `search()`, `findall()`, and `sub()`) under your belt, let's get into the details of building them. The patterns you make apply to any of these functions.

You've seen the basics:

- Literal matches with any nonspecial characters
- Any single character except `\n` with `.`
- Any number of the preceding character (including zero) with `*`
- Optional (zero or one) of the preceding character with `?`

First, Table 17-2 lists the special characters.

Table 17-2. Special characters

Pattern	Matches
`\d`	A single digit
`\D`	A single nondigit
`\w`	An alphanumeric character
`\W`	A non-alphanumeric character

Pattern	Matches
\s	A whitespace character
\S	A nonwhitespace character
\b	A word boundary (between a \w and a \W, in either order)
\B	A nonword boundary

The Python `string` module has predefined string constants that we can use for testing. Let's use `printable`, which contains 100 printable ASCII characters, including letters in both cases, digits, space characters, and punctuation:

```
>>> import string
>>> printable = string.printable
>>> len(printable)
100
>>> printable[0:50]
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN'
>>> printable[50:]
'OPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Which characters in `printable` are digits?

```
>>> re.findall('\d', printable)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Which characters are digits, letters, or an underscore?

```
>>> re.findall('\w', printable)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', '_']
```

Which are spaces?

```
>>> re.findall('\s', printable)
[' ', '\t', '\n', '\r', '\x0b', '\x0c']
```

In order, those are: plain old space, tab, newline, carriage return, vertical tab, and form feed.

Regular expressions are not confined to ASCII. A `\d` will match whatever Unicode calls a digit, not just ASCII characters `0` through `9`. Let's add two non-ASCII lowercase letters from FileFormat.info:

In this test, we'll throw in the following:

- Three ASCII letters
- Three punctuation symbols that should *not* match a `\w`

- A Unicode *LATIN SMALL LETTER E WITH CIRCUMFLEX* (\u00ea)
- A Unicode *LATIN SMALL LETTER E WITH BREVE* (\u0115)

```
>>> x = 'abc' + '-/*' + '\u00ea' + '\u0115'
```

As expected, this pattern finds only the letters:

```
>>> re.findall('\w', x)
['a', 'b', 'c', 'ê', 'ĕ']
```

Using Specifiers

Now let's make "punctuation pizza," using the main pattern specifiers for regular expressions, which are presented in Table 17-3. In the table, *expr* and the other italicized words mean any valid regular expression.

Table 17-3. Pattern specifiers

Pattern	Matches
abc	Literal abc
(*expr*)	*expr*
expr1 | *expr2*	*expr1* or *expr2*
.	Any character except \n
^	Start of source string
$	End of source string
prev ?	Zero or one *prev*
prev *	Zero or more *prev*, as many as possible
prev *?	Zero or more *prev*, as few as possible
prev +	One or more *prev*, as many as possible
prev +?	One or more *prev*, as few as possible
prev { *m* }	*m* consecutive *prev*
prev { *m*, *n* }	*m* to *n* consecutive *prev*, as many as possible
prev { *m*, *n* }?	*m* to *n* consecutive *prev*, as few as possible
[*abc*]	a or b or c (same as a|b|c)
[^ *abc*]	*not* (a or b or c)
prev (?= *next*)	*prev* if followed by *next*
prev (?! *next*)	*prev* if *not* followed by *next*
(?<= *prev*) *next*	*next* if preceded by *prev*
(?<! *prev*) *next*	*next* if *not* preceded by *prev*

Your eyes might cross permanently when trying to read these examples. First, let's define our source string:

```
>>> source = '''I wish I may, I wish I might
... Have a dish of fish tonight.'''
```

Now we apply different regular expression pattern strings to try to match something in the `source` string.

In the following examples, I use plain quoted strings for the patterns. A little later in this section, I show how a raw pattern string (`r` before the initial quote) helps avoid some conflicts between Python's normal string escapes and regular expression ones. So, to be safest, the first argument in all the following examples should actually be a raw string.

First, find `wish` anywhere:

```
>>> re.findall('wish', source)
['wish', 'wish']
```

Next, find `wish` or `fish` anywhere:

```
>>> re.findall('wish|fish', source)
['wish', 'wish', 'fish']
```

Find `wish` at the beginning:

```
>>> re.findall('^wish', source)
[]
```

Find `I wish` at the beginning:

```
>>> re.findall('^I wish', source)
['I wish']
```

Find `fish` at the end:

```
>>> re.findall('fish$', source)
[]
```

Finally, find `fish tonight.` at the end:

```
>>> re.findall('fish tonight.$', source)
['fish tonight.']
```

The characters `^` and `$` are called *anchors*: `^` anchors the search to the beginning of the search string, and `$` anchors it to the end. The `.$` characters match any character at the end of the line, including a period, so that works. To be more precise, we should escape the dot to match it literally:

```
>>> re.findall('fish tonight\.$', source)
['fish tonight.']
```

Begin by finding `w` or `f` followed by `ish`:

```
>>> re.findall('[wf]ish', source)
['wish', 'wish', 'fish']
```

Find one or more runs of `w`, `s`, or `h`:

```
>>> re.findall('[wsh]+', source)
['w', 'sh', 'w', 'sh', 'h', 'sh', 'sh', 'h']
```

Find `ght` followed by a non-alphanumeric:

```
>>> re.findall('ght\W', source)
['ght\n', 'ght.']
```

Find `I` followed by `wish`:

```
>>> re.findall('I (?=wish)', source)
['I ', 'I ']
```

And last, `wish` preceded by `I`:

```
>>> re.findall('(?<=I) wish', source)
[' wish', ' wish']
```

I mentioned earlier that in a few cases, the regular expression pattern rules conflict with the Python string rules. The following pattern should match any word that begins with `fish`:

```
>>> re.findall('\bfish', source)
[]
```

Why doesn't it? As is discussed in Chapter 4, Python employs a few special *escape characters* for strings. For example, `\b` means *backspace* in strings, but in the mini-language of regular expressions it means the *beginning* of a word. Avoid the accidental use of escape characters by using Python's *raw strings* when you define your regular expression string. Always put an `r` character before your regular expression pattern string, and Python escape characters will be disabled, as demonstrated here:

```
>>> re.findall(r'\bfish', source)
['fish']
```

Specifying match() Output

When using `match()` or `search()`, all matches are returned from the result object `m` as `m.group()`. If you enclose a pattern in parentheses, the match will be saved to its own group, and a tuple of them will be available as `m.groups()`, as shown here:

```
>>> m = re.search(r'(. dish\b).*(\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
```

Using the pattern `?P< name > expr` will match *expr*, saving the match in group *name*:

```
>>> m = re.search(r'(?P<DISH>. dish\b).*(?P<FISH>\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
>>> m.group('DISH')
'a dish'
>>> m.group('FISH')
'fish'
```

Review/Preview

This chapter covered some of the more advanced Python text topics: Unicode characters and regular expression pattern matching.

How can we top this? Binary data, the language that computers mutter to themselves when they think no one is looking.

Practice

17.1 Create a Unicode string called `mystery` and assign it the value `\U0001f984`. Print `mystery` and its Unicode name. Then assign `\U0001f4a9` to `mystery2` and do the same. (Your system font may or may not display an image for both.)

17.2 Encode `mystery`, this time using UTF-8, into the `bytes` variable `pop_bytes`. Print `pop_bytes`.

17.3 Using UTF-8, decode `pop_bytes` into the string variable `pop_string`. Print `pop_string`. Is `pop_string` equal to `mystery`?

17.4 When you're working with text, regular expressions come in handy. We'll apply them in numerous ways to our featured text sample. It's a poem titled "Ode on the Mammoth Cheese," written by James McIntyre in 1866 in homage to a seven-thousand-pound cheese that was crafted in Ontario and sent on an international tour. If you'd rather not type all of it, use your favorite search engine and cut and paste the words into your Python program, or just grab it from Project Gutenberg. Call the text string `mammoth`.

17.5 Import the `re` module to use Python's regular expression functions. Use `re.findall()` to print all the words that begin with `c`.

17.6 Find all four-letter words that begin with `c`.

17.7 Find all the words that end with `r`.

17.8 Find all the words that contain exactly three vowels in a row.

CHAPTER 18
Binary Data

Any sufficiently crisp question can be answered by a single binary digit—0 or 1, yes or no.
—Carl Sagan

Text data can be challenging, but binary data can be, well, interesting. You need to know about concepts such as *endianness* (the way your computer's processor breaks data into bytes) and *sign bits* for integers. You might need to delve into binary file formats or network packets to extract or even change data. This section shows you the basics of binary data wrangling in Python. You'll see how to read and write binary *files* in Chapter 20.

Convert Binary Data with struct

As you've seen, Python has many tools for manipulating text. Tools for binary data are much less prevalent. The standard library contains the `struct` module, which handles data similar to *structs* in C and C++. Using `struct`, you can convert binary data to and from Python data structures.

Let's see how this works with data from a PNG file, a common image format that you'll see along with GIF and JPEG files. We'll write a small program that extracts the width and height of an image from some PNG data.

We'll use the O'Reilly logo, the little bug-eyed tarsier shown in Figure 18-1.

Figure 18-1. The O'Reilly tarsier

The PNG file for this image is available on Wikipedia. I don't show how to read files until Chapter 20, so I downloaded this file, wrote a little program to print its values as bytes, and typed the values of the first 30 bytes into a Python `bytes` variable called `data` for the example that follows. (The PNG format specification says that the width and height are stored within the first 24 bytes, so we don't need more than that for now.)

Here's the code:

```
>>> import struct
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> if data[:8] == valid_png_header:
...     width, height = struct.unpack('>LL', data[16:24])
...     print('Valid PNG, width', width, 'height', height)
... else:
...     print('Not a valid PNG')
...
Valid PNG, width 154 height 141
```

Here's what this code does:

- `data` contains the first 30 bytes from the PNG file. To fit it on the page, I joined two byte strings with + and the continuation character (\).
- `valid_png_header` contains the eight-byte sequence that marks the start of a valid PNG file.
- `width` is extracted from bytes 16–19, and `height` from bytes 20–23.

The `>LL` is the format string that instructs `unpack()` how to interpret its input byte sequences and assemble them into Python data types. Here's the breakdown:

- The `>` means that integers are stored in *big-endian* format.
- Each `L` specifies a four-byte unsigned long integer.

You can examine each four-byte value directly:

```
>>> data[16:20]
b'\x00\x00\x00\x9a'
>>> data[20:24]0x9a
b'\x00\x00\x00\x8d'
```

Big-endian integers have the most significant bytes to the left. Because the width and height are each less than 255, they fit into the last byte of each sequence. You can verify that these hex values match the expected decimal values:

```
>>> 0x9a
154
>>> 0x8d
141
```

When you want to go in the other direction and convert Python data to bytes, use the `struct` `pack()` function:

```
>>> import struct
>>> struct.pack('>L', 154)
b'\x00\x00\x00\x9a'
>>> struct.pack('>L', 141)
b'\x00\x00\x00\x8d'
```

Tables 18-1 and 18-2 show the format specifiers for `pack()` and `unpack()`.

The endian specifiers go first in the format string.

Table 18-1. Endian specifiers

Specifier	Byte order
<	Little endian
>	Big endian

Table 18-2. Format specifiers

Specifier	Description	Bytes
x	Skip a byte	1
b	Signed byte	1
B	Unsigned byte	1
h	Signed short integer	2
H	Unsigned short integer	2

Specifier	Description	Bytes
`i`	Signed integer	4
`I`	Unsigned integer	4
`l`	Signed long integer	4
`L`	Unsigned long integer	4
`Q`	Unsigned long long integer	8
`f`	Single-precision float	4
`d`	Double-precision float	8
`p`	*count* and characters	1 + *count*
`s`	Characters	*count*

The type specifiers follow the endian character. Any specifier may be preceded by a number that indicates the *`count`*; `5B` is the same as `BBBBB`.

You can use a *`count`* prefix instead of `>LL`:

```
>>> struct.unpack('>2L', data[16:24])
(154, 141)
```

We used the slice `data[16:24]` to grab the interesting bytes directly. We could also use the `x` specifier to skip the uninteresting parts:

```
>>> struct.unpack('>16x2L6x', data)
(154, 141)
```

This means the following:

- Use big-endian integer format (`>`).
- Skip 16 bytes (`16x`).
- Read eight bytes—two unsigned long integers (`2L`).
- Skip the final six bytes (`6x`).

Extraction with Binary Data Tools

Some third-party open source packages offer the following, more declarative ways of defining and extracting binary data:

- bitstring
- Construct
- Hachoir
- Kaitai Struct

For the next example, you need to install Construct with `pip install construct`.

Here's how to extract the PNG dimensions from our `data` byte string by using Construct:

```
>>> from construct import Struct, Magic, UBInt32, Const, String
>>> # adapted from code at https://github.com/construct
>>> fmt = Struct('png',
...     Magic(b'\x89PNG\r\n\x1a\n'),
...     UBInt32('length'),
...     Const(String('type', 4), b'IHDR'),
...     UBInt32('width'),
...     UBInt32('height')
...     )
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> result = fmt.parse(data)
>>> print(result)
Container:
    length = 13
    type = b'IHDR'
    width = 154
    height = 141
>>> print(result.width, result.height)
154, 141
```

Convert Bytes/Strings with binascii()

The standard `binascii` module has functions to convert between binary data and various string representations: hex (base 16), base 64, uuencoded, and others. For example, in the next snippet, let's print that eight-byte PNG header as a sequence of hex values, instead of the mixture of ASCII and `\x` *xx* escapes that Python uses to display *bytes* variables:

```
>>> import binascii
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> print(binascii.hexlify(valid_png_header))
b'89504e470d0a1a0a'
```

Hey, this thing works backward too:

```
>>> print(binascii.unhexlify(b'89504e470d0a1a0a'))
b'\x89PNG\r\n\x1a\n'
```

Use Bit Operators

Python provides bit-level integer operators, similar to those in the C language. Table 18-3 summarizes them and includes examples with the integer variables `x` (decimal `5`, binary `0b0101`) and `y` (decimal `1`, binary `0b0001`).

Table 18-3. Bit-level integer operators

Operator	Description	Example	Decimal result	Binary result
`&`	And	`x & y`	`1`	`0b0001`
`|`	Or	`x | y`	`5`	`0b0101`
`^`	Exclusive or	`x ^ y`	`4`	`0b0100`
`~`	Flip bits	`~x`	`-6`	*Binary representation depends on int size*
`<<`	Left shift	`x << 1`	`10`	`0b1010`
`>>`	Right shift	`x >> 1`	`2`	`0b0010`

These operators work something like the set operators in Chapter 9. The `&` operator returns bits that are the same in both arguments, and `|` returns bits that are set in either of them. The `^` operator returns bits that are in one or the other, but not both. The `~` operator reverses all the bits in its single argument; this also reverses the sign because an integer's highest bit indicates its sign (`1` = negative) in *two's complement* arithmetic, used in all modern computers. The `<<` and `>>` operators just move bits to the left or right. A left shift of one bit is the same as multiplying by two, and a right shift is the same as dividing by two.

Review/Preview

This was all about trying to make sense of binary data.

The next chapter searches for order in the messy world of dates and times.

Practice

18.1 Use `unhexlify()` to convert this hex string (combined from two strings to fit on a page) to a `bytes` variable called `gif`:

```
'4749463839610100010080000000000fffffff21f9' +
'0401000000002c000000000100010000020144003b'
```

18.2 The bytes in `gif` define a 1-pixel transparent GIF file, one of the most common graphics file formats. A legal GIF starts with the string `GIF89a`. Does `gif` match this?

18.3 The pixel width of a GIF is a 16-bit little-endian integer starting at byte offset 6, and the height is the same size, starting at offset 8. Extract and print these values for `gif`. Are they both `1`?

CHAPTER 19
Dates and Times

"One!" strikes the clock in the belfry tower,
Which but sixty minutes ago
Sounded twelve for the midnight hour.

—Frederick B. Needham, "The Round of the Clock"

I've been on a calendar, but I have never been on time.

—Marilyn Monroe

Programmers devote a surprising amount of effort to dates and times. Let's talk about some of the problems they encounter and then explore some best practices and tricks to make the situation a little less messy.

Dates can be represented in many ways—too many ways, actually. Even in English with the Roman calendar, you'll see many variants of a simple date:

- July 21, 1987
- 21 Jul 1987
- 21/7/1987
- 7/21/1987

Among other problems, date representations can be ambiguous. In the previous examples, it's easy to determine that 7 stands for the month and 21 is the day of the month, because months don't go to 21. But how about 1/6/2012? Is that referring to January 6 or June 1? The United States uses month/day/year, but Europe assumes day/month/year. Canada commonly uses month/day/year in English, but day/month/year in French. Sacré bleu!

The month name varies by language within the Roman calendar. Even the year and month can have a different definition in other cultures.

Times have their own sources of grief, especially because of time zones and daylight saving time. If you look at a time-zone map, the zones follow political and historic boundaries rather than crisp lines every 15 degrees (360 degrees/24) of longitude. And countries start and end daylight saving times on different days of the year. Southern hemisphere countries advance their clocks as their northern friends are winding theirs back, and vice versa.

Python's standard library has many date and time modules, including: `datetime`, `time`, `calendar`, and others. There's some overlap, and it's a bit confusing.

Leap Year

Leap years are a special wrinkle in time. You probably know that every four years is a leap year (and the summer Olympics and the American presidential election). Did you also know that every 100 years is not a leap year, but that every 400 years is? Here's code to test various years for leapiness:

```
>>> import calendar
>>> calendar.isleap(1900)
False
>>> calendar.isleap(1996)
True
>>> calendar.isleap(1999)
False
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2002)
False
>>> calendar.isleap(2004)
True
```

For the curious:

- A year has 365.242196 days (after one spin around the sun, the earth is about a quarter-turn on its axis from where it started).
- Add one day every four years. Now an average year has 365.242196 – 0.25 = 364.992196 days
- Subtract a day every 100 years. Now an average year has 364.992196 + 0.01 = 365.002196 days
- Add a day every 400 years. Now an average year has 365.002196 – 0.0025 = 364.999696 days

Close enough for now! We will not speak of leap seconds.

The datetime Module

The standard `datetime` module handles (which should not be a surprise) dates and times. It defines four main object classes, each with many methods:

- `date` for years, months, and days
- `time` for hours, minutes, seconds, and fractions
- `datetime` for dates and times together
- `timedelta` for date and/or time intervals

You can make a `date` object by specifying a year, month, and day. Those values are then available as attributes:

```
>>> from datetime import date
>>> halloween = date(2025, 10, 31)
>>> halloween
datetime.date(2025, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2025
```

You can print a `date` with its `isoformat()` method:

```
>>> halloween.isoformat()
'2025-10-31'
```

The `iso` refers to ISO 8601, an international standard for representing dates and times. It goes from most general (year) to most specific (day). Because of this, it also sorts correctly: by year, then month, then day. I usually choose this format for date representation in programs, and for filenames that save data by date. The next section describes the more complex `strptime()` and `strftime()` methods for parsing and formatting dates.

This example uses the `today()` method to generate today's date:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2025, 2, 6)
```

This one makes use of a `timedelta` object to add a time interval to a `date`:

```
>>> from datetime import timedelta
>>> one_day = timedelta(days=1)
>>> tomorrow = now + one_day
>>> tomorrow
```

```
datetime.date(2025, 2, 7)
>>> now + 17*one_day
datetime.date(2025, 2, 23)
>>> yesterday = now - one_day
>>> yesterday
datetime.date(2025, 2, 5)
```

The range of `date` is from `date.min` (year=1, month=1, day=1) to `date.max` (year=9999, month=12, day=31). As a result, you can't use it for historic or astronomical calculations.

The `datetime` module's `time` object is used to represent a time of day:

```
>>> from datetime import time
>>> noon = time(12, 0, 0)
>>> noon
datetime.time(12, 0)
>>> noon.hour
12
>>> noon.minute
0
>>> noon.second
0
>>> noon.microsecond
0
```

The arguments go from the largest time unit (hours) to the smallest (microseconds). If you don't provide all the arguments, `time` assumes all the rest are zero. By the way, just because you can store and retrieve microseconds doesn't mean you can retrieve time from your computer to the exact microsecond. The accuracy of subsecond measurements depends on many factors in the hardware and operating system.

The `datetime` object includes both the date and time of day. You can create one directly, such as the one that follows, which is for January 2, 2025, at 3:04 a.m., plus 5 seconds and 6 microseconds:

```
>>> from datetime import datetime
>>> some_day = datetime(2025, 1, 2, 3, 4, 5, 6)
>>> some_day
datetime.datetime(2025, 1, 2, 3, 4, 5, 6)
```

The `datetime` object also has an `isoformat()` method:

```
>>> some_day.isoformat()
'2025-01-02T03:04:05.000006'
```

That middle `T` separates the date and time parts.

The `datetime` object has a `now()` method to return the current date and time:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
```

```
datetime.datetime(2025, 2, 6, 6, 40, 44, 536640)
>>> now.year
2025
>>> now.month
2
>>> now.day
6
>>> now.minute
40
>>> now.second
44
>>> now.microsecond
536640
```

You can `combine()` a `date` object and a `time` object into a `datetime`:

```
>>> from datetime import datetime, time, date
>>> noon = time(12)
>>> this_day = date.today()
>>> noon_today = datetime.combine(this_day, noon)
>>> noon_today
datetime.datetime(2025, 2, 6, 12, 0))
```

You can yank the `date` and `time` from a `datetime` by using the `date()` and `time()` methods:

```
>>> noon_today.date()
datetime.date(2025, 2, 6)
>>> noon_today.time()
datetime.time(12, 0)
```

The time Module

It is confusing that Python has a `datetime` module with a `time` object, and a separate `time` module. Furthermore, the `time` module has a function called—wait for it—`time()`.

One way to represent an absolute time is to count the number of seconds since a certain starting point. *Unix time* uses the number of seconds since midnight on January 1, 1970.[1] This value is often called the *epoch*, and it is often the simplest way to exchange dates and times among systems.

The `time` module's `time()` function returns the current time as an epoch value:

```
>>> import time
>>> now = time.time()
>>> now
1738845833.64127
```

1 This starting point is roughly when Unix was born, ignoring those pesky leap seconds.

Way more than one billion seconds have ticked by since New Year's, 1970. Where did the time go?

You can convert an epoch value to a string by using `ctime()`:

```
>>> time.ctime(now)
'Thu Feb  6 06:43:53 2025'
```

In the next section, you'll see how to produce more attractive formats for dates and times.

Epoch values are a useful least-common denominator for date and time exchange with different systems, such as JavaScript. Sometimes, though, you need actual days, hours, and so forth, which `time` provides as `struct_time` objects. The `localtime()` function provides the time in your system's time zone, and `gmtime()` provides it in UTC:

```
>>> time.localtime(now)
time.struct_time(tm_year=2025, tm_mon=2, tm_mday=6, tm_hour=6, tm_min=43,
tm_sec=53, tm_wday=3, tm_yday=37, tm_isdst=0)
>>> time.gmtime(now)
time.struct_time(tm_year=2025, tm_mon=2, tm_mday=6, tm_hour=12, tm_min=43,
tm_sec=53, tm_wday=3, tm_yday=37, tm_isdst=0)
```

My `06:43` (Central time zone, Daylight Saving) was `12:43` in UTC (formerly called *Greenwich time* or *Zulu time*). If you omit the argument to `localtime()` or `gmtime()`, they assume the current time.

Some of the `tm_...` values in `struct_time` are a bit ambiguous, so take a look at Table 19-1 for more details.

Table 19-1. `struct_time` values

Index	Name	Meaning	Values
0	`tm_year`	Year	`0000` to `9999`
1	`tm_mon`	Month	`1` to `12`
2	`tm_mday`	Day of month	`1` to `31`
3	`tm_hour`	Hour	`0` to `23`
4	`tm_min`	Minute	`0` to `59`
5	`tm_sec`	Second	`0` to `61`
6	`tm_wday`	Day of week	`0` (Monday) to `6` (Sunday)
7	`tm_yday`	Day of year	`1` to `366`
8	`tm_isdst`	Daylight saving?	`0` = no, `1` = yes, `-1` = unknown

If you don't want to type all those `tm_...` names, `struct_time` also acts like a named tuple (see "Named Tuples" on page 209), so you can use the indices from the previous table:

```
>>> import time
>>> now = time.localtime()
>>> now
time.struct_time(tm_year=2025, tm_mon=2, tm_mday=6, tm_hour=6, tm_min=46,
tm_sec=51, tm_wday=3, tm_yday=37, tm_isdst=0)
>>> now[0]
2025
>>> print(list(now[x] for x in range(9)))
[2025, 2, 6, 6, 46, 51, 3, 37, 0]
```

The `mktime()` function goes in the other direction, converting a `struct_time` object to epoch seconds:

```
>>> time.mktime(now)
1738846011.0
```

This doesn't exactly match our earlier epoch value of `now()` because the `struct_time` object preserves time only to the second.

Wherever possible, *use UTC* instead of time zones. UTC is an absolute time, independent of time zones. If you have a server, *set its time to UTC*; do not use local time.

In addition, *never use daylight saving time* if you can avoid it. If you use daylight saving time, an hour disappears at one time of year ("spring ahead") and occurs twice at another time ("fall back"). For some reason, many organizations use local time with daylight saving in their computer systems but are mystified twice every year by that spooky hour.

Read and Write Dates and Times

The `isoformat()` function is not the only way to write dates and times. You already saw the `ctime()` function in the `time` module, which you can use to convert epochs to strings:

```
>>> import time
>>> now = time.time()
>>> time.ctime(now)
'Thu Feb  6 06:50:31 2025'
```

You can also convert dates and times to strings by using `strftime()`. This is provided as a method in the `datetime`, `date`, and `time` objects, and as a function in the `time` module. The `strftime()` function uses format strings to specify the output, which you can see in Table 19-2.

Table 19-2. Output specifiers for `strftime()`

Format string	Date/time unit	Range
%Y	Year	1900–...
%m	Month	01–12
%B	Month name	January, ...
%b	Month abbrev	Jan, ...
%d	Day of month	01–31
%A	Weekday name	Sunday, ...
a	Weekday abbrev	Sun, ...
%H	Hour (24 hr)	00–23
%I	Hour (12 hr)	01–12
%p	AM/PM	AM, PM
%M	Minute	00–59
%S	Second	00–59

Numbers are zero-padded on the left.

Here's the `strftime()` function provided by the `time` module. It converts a `struct_time` object to a string. We'll first define the format string `fmt` and use it again later:

```
>>> import time
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> t = time.localtime()
>>> t
time.struct_time(tm_year=2025, tm_mon=2, tm_mday=6, tm_hour=6, tm_min=51,
tm_sec=11, tm_wday=3, tm_yday=37, tm_isdst=0)
>>> time.strftime(fmt, t)
"It's Thursday, February 06, 2025, local time 06:51:11AM"
```

If we try this with a `date` object, only the date parts will work, and the time defaults to midnight:

```
>>> from datetime import date
>>> some_day = date(2025, 2, 14)
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> some_day.strftime(fmt)
"It's Friday, February 14, 2025, local time 12:00:00AM""
```

For a `time` object, only the time parts are converted:

```
>>> from datetime import time
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> some_time = time(10, 35)
>>> some_time.strftime(fmt)
"It's Monday, January 01, 1900, local time 10:35:00AM"
```

You won't want to use the day parts from a `time` object, because they're meaningless.

To go the other way and convert a string to a date or time, use `strptime()` with the same format string. No regular expression pattern matching occurs; the nonformat parts of the string (without `%`) need to match exactly. Let's specify a format that matches *year-month-day*, such as `2025-02-14`. What happens if the date string you want to parse has spaces instead of dashes?

```
>>> # (modified with ... to save print space)
>>> import time
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2025 02 14", fmt)
Traceback (most recent call last):
  File "<python-input-58>", line 1, in <module>
    time.strptime("2025 02 14", fmt)
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^
  File ".../python3.13/_strptime.py", line 567, in _strptime_time
    tt = _strptime(data_string, format)[0]
         ~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^
  File ".../python3.13/_strptime.py", line 352, in _strptime
    raise ValueError("time data %r does not match format %r" %
                     (data_string, format))
ValueError: time data '2025 02 14' does not match format '%Y-%m-%d
```

If we feed `strptime()` some dashes, is it happy now?

```
>>> import time
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2025-02-14", fmt)
time.struct_time(tm_year=2025, tm_mon=2, tm_mday=14, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=4, tm_yday=45, tm_isdst=-1)
```

Or we can fix the `fmt` string to match the date string:

```
>>> import time
>>> fmt = "%Y %m %d"
>>> time.strptime("2025 02 14", fmt)
time.struct_time(tm_year=2025, tm_mon=2, tm_mday=14, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=4, tm_yday=45, tm_isdst=-1)
```

Even if the string seems to match its format, an exception is raised if a value is out of range (filenames truncated for space):

```
>>> fmt = "%Y-%m-%d"
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2025-02-30", fmt)
Traceback (most recent call last):
  File "<python-input-66>", line 1, in <module>
    time.strptime("2025-02-30", fmt)
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^
  File ".../python3.13/_strptime.py",
    line 567, in _strptime_time
    tt = _strptime(data_string, format)[0]
```

```
        ~~~~~~~~~^^^^^^^^^^^^^^^^^^^^
  File ".../python3.13/_strptime.py",
    line 539, in _strptime
    julian = datetime_date(year, month, day).toordinal() - \
             ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^
ValueError: day is out of range for month
```

Names are specific to your *locale*—internationalization settings for your operating system. If you need to print different month and day names, change your locale by using `setlocale()`; its first argument is `locale.LC_TIME` for dates and times, and the second is a string combining the language and country abbreviation.

Let's invite some international friends to a Halloween party. We'll print the day of the week, month, and day in US English, French, German, Spanish, and Icelandic (Icelanders have real elves):

```
>>> import locale
>>> from datetime import date
>>> halloween = date(2025, 10, 31)
>>> for lang_country in ['en_us', 'fr_fr', 'de_de', 'es_es', 'is_is',]:
...     locale.setlocale(locale.LC_TIME, lang_country)
...     halloween.strftime('%A, %B %d')
...
'en_us'
'Friday, October 31'
'fr_fr'
'Vendredi, octobre 31'
'de_de'
'Freitag, Oktober 31'
'es_es'
'viernes, octubre 31'
'is_is'
'föstudagur, október 31'
>>>
```

Where do you find these magic values for `lang_country`? The following code is a bit wonky, but you can try this to get all of them (there are a few hundred):

```
>>> import locale
>>> names = locale.locale_alias.keys()
```

From `names`, let's get just locale names that seem to work with `setlocale()`, such as the ones we used in the preceding example—a two-character language code followed by an underscore and a two-character country code:

```
>>> good_names = [name for name in names if
len(name) == 5 and name[2] == '_']
```

What do the first five look like?

```
>>> good_names[:5]
['a3_az', 'aa_dj', 'aa_er', 'aa_et', 'af_za']
```

So, if you want all the German language locales, try this:

```
>>> de = [name for name in good_names if name.startswith('de')]
>>> de
['de_at', 'de_be', 'de_ch', 'de_de', 'de_it', 'de_lu']
```

If you run `set_locale()` and get the error

```
locale.Error: unsupported locale setting
```

that locale is not supported by your operating system. You'll need to figure out what your operating system needs to add the locale. This error can happen even if Python tells you (using `locale.locale_alias.keys()`) that it is a good locale. I had this error when testing on macOS with the locale `cy_gb` (Welsh, Great Britain), even though it had accepted `is_is` (Icelandic) in the preceding example.

All the Conversions

Figure 19-1 (from the Python wiki) summarizes all the standard Python time interconversions.

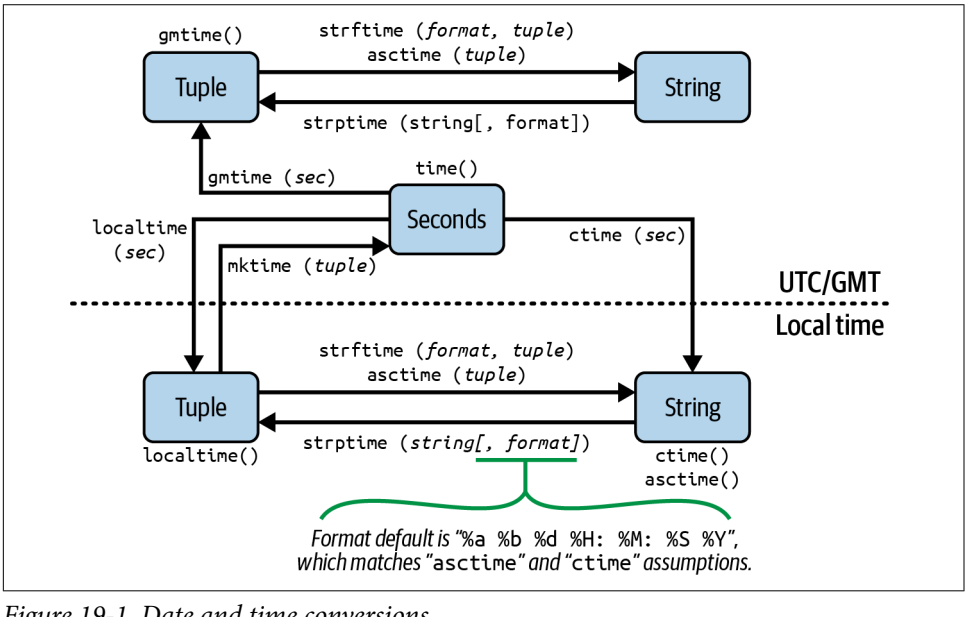

Figure 19-1. Date and time conversions

Alternative Modules

If you find the standard library modules confusing or lacking a particular conversion that you want, many third-party alternatives are available. Here are just a few:

Whenever
: A new, fast library, written in Rust

Arrow
: Combines many date and time functions with a simple API

dateutil
: Parses almost any date format and handles relative dates and times well

iso8601
: Fills in gaps in the standard library for the ISO8601 format

Fleming
: Many time-zone functions

Maya
: Intuitive interface to dates, times, and intervals

dateinfer
: Guesses the right format strings from date/time strings; no longer modified

If you need *really* big dates, as in astronomy, see one of these:

Astropy
: Has many modules for astronomical data, including `astro.time`.

apwlib
: Has a `datetime` subclass called `astrodatetime`.

TAI64
: *TAI* (from the French for International Atomic Time) uses 64 bits to express a date range of 292 billion years, in second or nanosecond resolution.

Review/Preview

Clocks and calendars: it's surprising how often you need to deal with their oddities.

The coming chapter deals with good old files: how to open, read, write, rename, and other imaginative pastimes.

Practice

19.1 Write the current date as a string to the text file *today.txt*.

19.2 Read the text file *today.txt* into the string `today_string`.

19.3 Parse the date from `today_string`.

19.4 Create a date object of your day of birth.

19.5 What day of the week was your day of birth?

19.6 When will you be (or when were you) 10,000 days old?

CHAPTER 20
Files

We have persistent objects. They're called files.
—Ken Thompson

When you first start programming, you hear some words over and over but aren't sure whether they have a specific technical meaning or are just hand-waving. The terms *file* and *directory* are such words, and they do have actual technical meanings. A *file* is a sequence of bytes, stored in a *filesystem*, and accessed by a *filename*. A *directory* is a collection of files and possibly other directories. The term *folder* is a synonym for *directory*. It turned up when computers gained graphical user interfaces, and mimicked office concepts to make concepts seem more familiar.

Many filesystems are hierarchical and often referred to as being like a tree. Real offices don't tend to have trees, and the folder analogy works only if you visualize subfolders all the way down.

File Input and Output

The simplest kind of persistence is a plain old file, sometimes called a *flat file*. You *read* from a file into memory and *write* from memory to a file. Python makes these jobs easy. As with many languages, its file operations were largely modeled on the familiar and popular Unix equivalents.

Create or Open with open()

You need to call the `open()` function before you do the following:

- Read an existing file
- Write to a new file
- Append to an existing file
- Overwrite an existing file

Here's a brief explanation of the pieces of this call:

```
fileobj = open( filename, mode )
```

- *`fileobj`* is the file object returned by `open()`.
- *`filename`* is the string name of the file.
- *`mode`* is a string indicating the file's type and what you want to do with it.

The first letter of *`mode`* indicates the *operation*:

- `r` means *read*.
- `w` means *write*. If the file doesn't exist, it's created. If the file does exist, it's overwritten.
- `x` means *write*, but only if the file does *not* already exist.
- `a` means *append* (write after the end) if the file exists.

The second letter of *`mode`* is the file's *type*:

- `t` (or nothing) means *text*.
- `b` means *binary*.

After opening the file, you call functions to read or write data; these will be shown in the examples that follow.

Last, you need to *close* the file to ensure that any writes complete and that memory is freed. Later, you'll see how to use `with` to automate this for you.

This program opens a file called *oops.txt* and closes it without writing anything. This creates an empty file:

```
>>> fout = open('oops.txt', 'wt')
>>> fout.close()
```

Write a Text File with print()

Let's re-create *oops.txt*, but now write a line to it and then close it:

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

We created an empty *oops.txt* file in the previous section, so this just overwrites it.

We use the `file` argument to `print()`. Without it, `print()` writes to *standard output*, which is your terminal (unless you've told your shell program to redirect output to a file with `>` or piped it to another program with `|`).

Write a Text File with write()

We just used `print()` to write a line to a file. We can also use `write()`.

For our multiline data source, let's use this limerick about special relativity:[1]

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
```

The following code writes the entire poem to the file *relativity* in one call:

```
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
150
>>> fout.close()
```

The `write` function returns the number of bytes written. It does not add any spaces or newlines, as `print()` does. As before, you can also `print` a multiline string to a text file:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

So, should you use `write()` or `print()`? As you've seen, by default `print()` adds a space after each argument and a newline at the end. In the previous example, it appended a newline to the *relativity* file.

1 In the first manuscript of this book, I said *general* relativity and was kindly corrected by a physicist reviewer.

To make `print()` work like `write()`, pass it the following two arguments:

- `sep` (separator, which defaults to a space, `' '`)
- `end` (end string, which defaults to a newline, `\n`)

We'll use empty strings to replace these defaults:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

If you have a large source string, you can also write chunks (using slices) until the source is done:

```
>>> fout = open('relativity', 'wt')
>>> size = len(poem)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(poem[offset:offset+chunk])
...     offset += chunk
...
100
50
>>> fout.close()
```

This writes 100 characters on the first try and the last 50 characters on the next. Slices allow you to "go over the end" without raising an exception.

If the *relativity* file is precious to us, let's see whether using mode `x` really protects us from overwriting it:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

You can use this `x` mode check with an exception handler:

```
>>> try:
...     fout = open('relativity', 'xt')]
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close one.')
...
relativity already exists!. That was a close one.
```

Read a Text File with read(), readline(), or readlines()

You can call `read()` with no arguments to slurp up the entire file at once, as shown in the example that follows (be careful when doing this with large files; a gigabyte file will consume a gigabyte of memory):

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

You can provide a maximum character count to limit the amount that `read()` returns at one time. Let's read 100 characters at a time and append each chunk to a `poem` string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

After you've read all the way to the end, further calls to `read()` will return an empty string (`''`), which is treated as `False` in `if not fragment`. This breaks out of the `while True` loop.

You can also read the file a line at a time by using `readline()`. In this next example, we append each line to the `poem` string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
...     line = fin.readline()
...     if not line:
...         break
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

For a text file, even a blank line has a length of one (the newline character) and is evaluated as `True`. When the file has been read, `readline()`, like `read()`, also returns an empty string, which is also evaluated as `False`.

The easiest way to read a text file is by using an *iterator*. This returns one line at a time. It's similar to the previous example but with less code:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

All the preceding examples eventually build the single string `poem`. The `readlines()` call reads a line at a time and returns a list of one-line strings:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
...     print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

We told `print()` to suppress the automatic newlines because the first four lines already had them. The last line did not, causing the interactive prompt `>>>` to occur right after the last line.

Write a Binary File with write()

If you include a `b` in the *mode* string, the file is opened in binary mode. In this case, you read and write `bytes` instead of a string.

We don't have a binary poem lying around and don't want to ask ChatGPT, so we'll just generate the 256 byte values from 0 to 255:

```
>>> bdata = bytes(range(0, 256))
>>> len(bdata)
256
```

Open the file for writing in binary mode and write all the data at once:

```
>>> fout = open('bfile', 'wb')
>>> fout.write(bdata)
256
>>> fout.close()
```

Again, `write()` returns the number of bytes written.

As with text, you can write binary data in chunks:

```
>>> fout = open('bfile', 'wb')
>>> size = len(bdata)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(bdata[offset:offset+chunk])
...     offset += chunk
...
100
100
56
>>> fout.close()
```

Read a Binary File with read()

Reading a binary file is simple; all you need to do is just open with `rb`:

```
>>> fin = open('bfile', 'rb')
>>> bdata = fin.read()
>>> len(bdata)
256
>>> fin.close()
```

Close Files Automatically by Using with

If you forget to close a file that you've opened, it will be closed by Python after it's no longer referenced. This means that if you open a file within a function and don't close it explicitly, it will be closed automatically when the function ends. But you might have opened the file in a long-running function or the main section of the program. The file should be closed to force any remaining writes to be completed.

Python has *context managers* to clean up issues such as open files. You use the form `with` *`expression`* `as` *`variable`*:

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
...
```

That's it. After the block of code under the context manager (in this case, one line) completes (normally *or* by a raised exception), the file is closed automatically.

Change Position with seek()

As you read and write, Python keeps track of where you are in the file. The `tell()` function returns your current offset from the beginning of the file, in bytes. The

`seek()` function lets you jump to another byte offset in the file. This means that you don't have to read every byte in a file to read the last one; you can `seek()` to the last one and read just one byte.

For this example, use the 256-byte binary file *bfile* that you wrote earlier:

```
>>> fin = open('bfile', 'rb')
>>> fin.tell()
0
```

Use `seek()` to jump to one byte before the end of the file:

```
>>> fin.seek(255)
255
```

Read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

The `seek()` function also returns the current offset.

You can call `seek()` with a second argument: `seek(`*`offset, origin`*`)`:

- If *`origin`* is `0` (the default), go *`offset`* bytes from the start.
- If *`origin`* is `1`, go *`offset`* bytes from the current position.
- If *`origin`* is `2`, go *`offset`* bytes relative to the end.

These values are also defined in the standard `os` module:

```
>>> import os
>>> os.SEEK_SET
0
>>> os.SEEK_CUR
1
>>> os.SEEK_END
2
```

So, we could have read the last byte in different ways:

```
>>> fin = open('bfile', 'rb')
```

Here, we read one byte before the end of the file:

```
>>> fin.seek(-1, 2)
255
>>> fin.tell()
255
```

Read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

You don't need to call `tell()` for `seek()` to work. I just want to show that they both report the same offset.

Here's an example of seeking from the current position in the file:

```
>>> fin = open('bfile', 'rb')
```

This next example ends up two bytes before the end of the file:

```
>>> fin.seek(254, 0)
254
>>> fin.tell()
254
```

Now go forward one byte:

```
>>> fin.seek(1, 1)
255
>>> fin.tell()
255
```

Finally, read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

These functions are most useful for binary files. You can use the functions with text files, but unless the file is ASCII (one byte per character), you would have a hard time calculating offsets. These would depend on the text encoding, and the most popular encoding (UTF-8) uses varying numbers of bytes per character.

Memory Mapping

An alternative to reading and writing a file is to *memory-map* it with the standard `mmap` module. This makes the contents of a file look like a `bytearray` in memory. See the documentation and some examples for more details.

File Operations

Python, like many other languages, patterns its file operations after Unix. Some functions, such as `chown()` and `chmod()`, have the same names, but there are a few new ones.

I'll first show how Python handles these tasks with functions from the `os.path` module and then with the newer `pathlib` module.

Check Existence with exists()

To verify whether the file or directory is really there or you just imagined it, you can provide `exists()`, with a relative or absolute pathname, as demonstrated here:

```
>>> import os
>>> os.path.exists('oops.txt')
True
>>> os.path.exists('./oops.txt')
True
>>> os.path.exists('waffles')
False
>>> os.path.exists('.')
True
>>> os.path.exists('..')
True
```

Check Type with isfile()

The functions in this section check whether a name refers to a file, directory, or symbolic link (see the examples that follow for a discussion of links).

The first function we'll look at, `isfile()`, asks a simple question: is it a plain old law-abiding file?

```
>>> name = 'oops.txt'
>>> os.path.isfile(name)
True
```

Here's how you determine a directory:

```
>>> os.path.isdir(name)
False
```

A single dot (`.`) is shorthand for the current directory, and two dots (`..`) stands for the parent directory. These always exist, so a statement such as the following will always report `True`:

```
>>> os.path.isdir('.')
True
```

The os module contains many functions dealing with *pathnames* (fully qualified filenames, starting with / and including all parents). One such function, isabs(), determines whether its argument is an absolute pathname. The argument doesn't need to be the name of a real file:

```
>>> os.path.isabs(name)
False
>>> os.path.isabs('/big/fake/name')
True
>>> os.path.isabs('big/fake/name/without/a/leading/slash')
False
```

Copy with copy()

The copy() function comes from another module, shutil. This example copies the file *oops.txt* to the file *ohno.txt*:

```
>>> import shutil
>>> shutil.copy('oops.txt', 'ohno.txt')
```

The shutil.move() function copies a file and then removes the original.

Change Name with rename()

This function does exactly what it says. In the example here, it renames *ohno.txt* to *ohwell.txt*:

```
>>> import os
>>> os.rename('ohno.txt', 'ohwell.txt')
```

Link with link() or symlink()

In Unix, a file exists in one place, but it can have multiple names, called *links*. In low-level *hard links*, it's not easy to find all the names for a given file. A *symbolic link* is an alternative method that stores the new name as its own file, making it possible for you to get both the original and new names at once. The link() call creates a hard link, and symlink() makes a symbolic link. The islink() function checks whether the file is a symbolic link.

Here's how to make a hard link to the existing file *oops.txt* from the new file *yikes.txt*:

```
>>> os.link('oops.txt', 'yikes.txt')
>>> os.path.isfile('yikes.txt')
True
>>> os.path.islink('yikes.txt')
False
```

To create a symbolic link to the existing file *oops.txt* from the new file *jeepers.txt*, use the following:

```
>>> os.symlink('oops.txt', 'jeepers.txt')
>>> os.path.islink('jeepers.txt')
True
```

Which should you use? Hard links might be useful for rare system tasks like file deduplication. I've always used symbolic links.

Change Permissions with chmod()

On a Unix system, `chmod()` changes file permissions. There are read, write, and execute permissions for the user (that's usually you, if you created the file), the main group that the user is in, and the rest of the world. The command takes an intensely compressed octal (base 8) value that combines user, group, and other permissions. For instance, to make *oops.txt* readable only by its owner, type the following:

```
>>> os.chmod('oops.txt', 0o400)
```

If you don't want to deal with cryptic octal values and would rather deal with (slightly less) obscure cryptic symbols, you can import some constants from the `stat` module and use a statement such as the following:

```
>>> import stat
>>> os.chmod('oops.txt', stat.S_IRUSR)
```

Change Ownership with chown()

This function is also Unix/Linux/Mac–specific. You can change the owner and/or group ownership of a file by specifying the numeric user ID (`uid`) and group ID (`gid`):

```
>>> uid = 5
>>> gid = 22
>>> os.chown('oops', uid, gid)
```

Delete a File with remove()

In this snippet, we use the `remove()` function and say farewell to *oops.txt*:

```
>>> os.remove('oops.txt')
>>> os.path.exists('oops.txt')
False
```

Directory Operations

In most operating systems, files exist in a hierarchy of *directories* (often called *folders*). The container of all these files and directories is a *filesystem* (sometimes called a *volume*). The standard `os` module deals with operating specifics such as these and provides the following functions with which you can manipulate them.

Create with mkdir()

This example shows how to create a directory called `poems` to store that precious verse:

```
>>> os.mkdir('poems')
>>> os.path.exists('poems')
True
```

Delete with rmdir()

Upon second thought,[2] you decide you don't need that directory after all. Here's how to delete it:

```
>>> os.rmdir('poems')
>>> os.path.exists('poems')
False
```

List Contents with listdir()

OK, take two: let's make `poems` again, with some contents:

```
>>> os.mkdir('poems')
```

Now get a list of its contents (none so far):

```
>>> os.listdir('poems')
[]
```

Next, make a subdirectory:

```
>>> os.mkdir('poems/mcintyre')
>>> os.listdir('poems')
['mcintyre']
```

Create a file in this subdirectory (don't type all these lines unless you really feel poetic; just make sure you begin and end with matching quotes, either single or tripled):

```
>>> fout = open('poems/mcintyre/the_good_man', 'wt')
>>> fout.write('''Cheerful and happy was his mood,
... He to the poor was kind and good,
... And he oft' times did find them food,
... Also supplies of coal and wood,
... He never spake a word was rude,
... And cheer'd those did o'er sorrows brood,
... He passed away not understood,
... Because no poet in his lays
... Had penned a sonnet in his praise,
... 'Tis sad, but such is world's ways.
... ''')
```

2 Why is it never first?

```
344
>>> fout.close()
```

Finally, let's see what we have. That new file had better be there:

```
>>> os.listdir('poems/mcintyre')
['the_good_man']
```

Change Current Directory with chdir()

With this function, you can go from one directory to another. Let's leave the current directory and spend a little time in `poems`:

```
>>> import os
>>> os.chdir('poems')
>>> os.listdir('.')
['mcintyre']
```

List Matching Files with glob()

The `glob()` function matches file or directory names by using Unix shell rules rather than the more complete regular expression syntax. Here are those rules:

- `*` matches everything (`re` would expect `.*`).
- `?` matches a single character.
- `[abc]` matches character `a`, `b`, or `c`.
- `[!abc]` matches any character *except* `a`, `b`, or `c`.

Try getting all files or directories that begin with `m`:

```
>>> import glob
>>> glob.glob('m*')
['mcintyre']
```

How about any two-letter files or directories?

```
>>> glob.glob('??')
[]
```

I'm thinking of an eight-letter word that begins with `m` and ends with `e`:

```
>>> glob.glob('m??????e')
['mcintyre']
```

What about anything that begins with a `k`, `l`, or `m`, and ends with `e`?

```
>>> glob.glob('[klm]*e')
['mcintyre']
```

Pathnames

Almost all our computers use hierarchical filesystems, with directories containing files and other directories, down to various levels. When you want to refer to a specific file or directory, you need its *pathname*: the sequence of directories needed to get there, either *absolute* from the top (the *root*), or *relative* to your current directory.

You'll often hear people confusing a forward *slash* (/, not the Guns N' Roses guy) and *backslash* (\).[3] Unix and Macs (and web URLs) use a slash as the *path separator*, and Windows uses a backslash.[4]

Python lets you use a slash as the path separator when you're specifying names. On Windows, you can use a backslash, but you know that a backslash is a ubiquitous escape character in Python, so you have to double it everywhere or use Python's raw strings:

```
>>> win_file = 'eek\\urk\\snort.txt'
>>> win_file2 = r'eek\urk\snort.txt'
>>> win_file
'eek\\urk\\snort.txt'
>>> win_file2
'eek\\urk\\snort.txt'
```

When you're building a pathname, you can do the following:

- Use the appropriate path separation character (`'/'` or `'\'`).
- Build a pathname (see "Build a Pathname with os.path.join()" on page 344).
- Use pathlib (see "Use pathlib" on page 344).

Get a Pathname with abspath()

This function expands a relative name to an absolute one. If your current directory is */usr/gaberlunzie* and the file *oops.txt* is there, you can type the following:

```
>>> os.path.abspath('oops.txt')
'/usr/gaberlunzie/oops.txt'
```

3 One way to remember: a forward slash tilts *forward*, a backslash tilts *back*.

4 QDOS was the operating system that Bill Gates bought for $50,000 to have "MS-DOS" when IBM came calling about their first PC. It mimicked CP/M, which used slashes for command-line arguments. When MS-DOS later added folders, it had to use backslashes.

Get a symlink Pathname with realpath()

In one of the earlier sections, we made a symbolic link to *oops.txt* from the new file *jeepers.txt*. In circumstances such as this, you can get the name of *oops.txt* from *jeepers.txt* by using the `realpath()` function, as shown here:

```
>>> os.path.realpath('jeepers.txt')
'/usr/gaberlunzie/oops.txt'
```

Build a Pathname with os.path.join()

When you're constructing a multipart pathname, you can call `os.path.join()` to combine them pairwise with the proper path separation character for your operating system:

```
>>> import os
>>> win_file = os.path.join("eek", "urk")
>>> win_file = os.path.join(win_file, "snort.txt")
```

If I run this on a Mac or Linux box, I get this:

```
>>> win_file
'eek/urk/snort.txt'
```

Running on Windows would produce this:

```
>>> win_file
'eek\\urk\\snort.txt'
```

But if the same code produces a different result depending on where it's run, that could be a problem. The new `pathlib` module is a portable solution to this.

Use pathlib

Python added the `pathlib` module in version 3.4. It's an alternative to the `os.path` modules that I just described. But why do we need another module?

Rather that treating filesystem pathnames as strings, it introduces the `Path` object to treat them at a little higher level. Create a `Path` with the `Path()` class, and then knit your path together with bare slashes (not `/` characters):

```
>>> from pathlib import Path
>>> file_path = Path('eek') / 'urk' / 'snort.txt'
>>> file_path
PosixPath('eek/urk/snort.txt')
>>> print(file_path)
eek/urk/snort.txt
```

This slash trick takes advantage of Python's magic methods (see "Magic Methods" on page 205). A `Path` can tell you a bit about itself:

```
>>> file_path.name
'snort.txt'
>>> file_path.suffix
'.txt'
>>> file_path.stem
'snort'
```

You can feed `file_path` to `open()` as you would any filename or pathname string.

You can also see what would happen if you ran this program on another system or if you needed to generate foreign pathnames on your computer:

```
>>> from pathlib import PureWindowsPath
>>> PureWindowsPath(file_path)
PureWindowsPath('eek/urk/snort.txt')
>>> print(PureWindowsPath(file_path))
eek\urk\snort.txt
```

See the docs for all the details.

BytesIO and StringIO

You've seen how to modify data in memory and how to get data in and out of files. What do you do if you have in-memory data but want to call a function that expects a file (or the reverse)? You'd want to modify the data and pass those bytes or characters around, without reading and writing temporary files.

You can use `io.BytesIO` for binary data (`bytes`) and `io.StringIO` for text data (`str`). Using either of these wraps data as a *file-like object*, suitable to use with all the file functions you've seen in this chapter.

One use case for this is data format conversion. Let's apply this to the Pillow library, which reads and writes image data. The first argument to its `Image` object's `open()` and `save()` methods is a filename *or a file-like object*. The code in Example 20-1 uses `BytesIO` to read *and* write in-memory data. It reads one or more image files from the command line, converts its image data to three formats, and prints the length and first 10 bytes of these outputs.

Example 20-1. convert_image.py

```
from io import BytesIO
from PIL import Image
import sys

def data_to_img(data):
    """Return PIL Image object, with data from in-memory <data>"""
```

```
    fp = BytesIO(data)
    return Image.open(fp)    # reads from memory

def img_to_data(img, fmt=None):
    """Return image data from PIL Image <img>, in <fmt> format"""
    fp = BytesIO()
    if not fmt:
        fmt = img.format     # keeps the original format
    img.save(fp, fmt)        # writes to memory
    return fp.getvalue()

def convert_image(data, fmt=None):
    """Convert image <data> to PIL <fmt> image data"""
    img = data_to_img(data)
    return img_to_data(img, fmt)

def get_file_data(name):
    """Return PIL Image object for image file <name>"""
    img = Image.open(name)
    print("img", img, img.format)
    return img_to_data(img)

if __name__ == "__main__":
    for name in sys.argv[1:]:
        data = get_file_data(name)
        print("in", len(data), data[:10])
        for fmt in ("gif", "png", "jpeg"):
            out_data = convert_image(data, fmt)
            print("out", len(out_data), out_data[:10])
```

Because it acts like a file, you can `seek()`, `read()`, and `write()` a `BytesIO` object just like a normal file; if you did a `seek()` followed by a `read()`, you would get only the bytes from that seek position to the end. That `getvalue()` function returns all the bytes in the `BytesIO` object.

Here's the output, using an input image file:

```
$ python convert_image.py ch20_critter.png
img <PIL.PngImagePlugin.PngImageFile image mode=RGB
  size=154x141 at 0x10340CF28> PNG
in 24941 b'\\x89PNG\\r\\n\\x1a\\n\\x00\\x00'
out 14751 b'GIF87a\\x9a\\x00\\x8d\\x00'
out 24941 b'\\x89PNG\\r\\n\\x1a\\n\\x00\\x00'
out 5914 b'\\xff\xd8\\xff\\xe0\\x00\\x10JFIF'
```

File Formats: Determination

> *You may tell yourself: this is not my beautiful file.*
>
> —apologies to The Talking Heads

Often you'll come upon a file and wonder just what it is, and maybe more importantly, what it might do. You'll read more about text data in Chapter 17 and binary data in Chapter 18. But just for now, are there easy ways to determine what the file does, without performing the forensic procedures that I described in the previous sections? Yes.

- The `file` program, which has been in Unix and its successors like Linux and macOS for a long time, identifies files by various "magic" byte sequences in particular file locations.
- This is based on the libmagic library, which the python-magic package interfaces for you.
- You can find many online file guessers and converters with Google.
- But, speaking of Google, it recently released an AI-based file type guesser called Magika. It's used in production there to route files to specific security scanners. `pip install magika` gets you the command-line tool and Python package.

Review/Preview

This chapter covered the management of files and directories: naming, ownership, reading, writing, and so on.

The next chapter gets into the tricky topic of concurrency.

Practice

20.1 List the files in your current directory.

20.2 List the files in your parent directory.

20.3 Assign the string `This is a test of the emergency text system` to the variable `test1`, and write `test1` to a file called *test.txt*.

20.4 Open the file *test.txt* and read its contents into the string `test2`. Are `test1` and `test2` the same?

CHAPTER 21

Data in Time: Concurrency

All we have to decide is what to do with the time that is given us.
—J. R. R. Tolkien

This chapter and the next two are longer and a bit more challenging than earlier ones: data in time (sequential and concurrent access on a single computer), then data in space (networking), and finally data in a box (storage and retrieval with special files and databases).

Programs and Processes

When you run an individual program, your operating system creates a single *process*. It uses system resources (CPU, memory, disk space) and data structures in the operating system's *kernel* (file and network connections, usage statistics, and so on). A process is isolated from other processes; it can't see what other processes are doing or interfere with them.

The operating system keeps track of all the running processes, giving each a little time to run and then switching to another, with the twin goals of spreading the work around fairly and being responsive to the user. You can see the state of your processes with graphical interfaces such as the Mac's Activity Monitor (macOS), Task Manager on Windows-based computers, or the `top` command in Linux.

You can also access process data from your own programs. The standard library's `os` module provides a common way of accessing some system information. For instance, the following functions get the *process ID* and the *current working directory* of the running Python interpreter:

```
>>> import os
>>> os.getpid()
76051
>>> os.getcwd()
'/Users/williamlubanovic'
```

And these get my *user ID* and *group ID*:

```
>>> os.getuid()
501
>>> os.getgid()
20
```

Create a Process with subprocess

All the programs that you've seen here so far have been individual processes. You can start and stop other existing programs from Python by using the standard library's `subprocess` module. If you just want to run another program in a shell and grab whatever output it created (both standard output and standard error output), use the `getoutput()` function. Here, we get the output of the Unix `date` program:

```
>>> import subprocess
>>> ret = subprocess.getoutput('date')
>>> ret
'Thu Jul 31 04:15:51 PM CDT 2025'
```

You won't get anything back until the process ends. If you need to call something that might take a lot of time, see "Concurrency" on page 356. Because the argument to `getoutput()` is a string representing a complete shell command, you can include arguments, pipes, < and > I/O redirection, and so on:

```
>>> ret = subprocess.getoutput('date -u')
>>> ret
'Thu Jul 31 09:16:30 PM UTC 2025'
```

Piping that output string to the `wc` command counts one line, seven "words," and 32 characters:

```
>>> ret = subprocess.getoutput('date -u | wc')
>>> ret
'      1       7      32'
```

A variant method called `check_output()` takes a list of the command and arguments. By default it returns standard output only as type bytes rather than a string, and does not use the shell:

```
>>> ret = subprocess.check_output(['date', '-u'])
>>> ret
b'Thu Jul 31 09:18:00 PM UTC 2025\n'
```

To show the exit status of the other program, `getstatusoutput()` returns a tuple with the status code and output:

```
>>> ret = subprocess.getstatusoutput('date')
>>> ret
(0, 'Thu Jul 31 04:18:37 PM CDT 2025')
```

If you don't want to capture the output but might want to know its exit status, use `call()`:

```
>>> ret = subprocess.call('date')
Thu Jul 31 04:19:11 PM CDT 2025
>>> ret
0
```

(In Unix-like systems, `0` is usually the exit status for success.)

That date and time is printed to output but not captured within our program. So, we save the return code as `ret`.

You can run programs with arguments in two ways. The first is to specify them in a single string. Our sample command is `date -u`, which prints the current date and time in UTC:

```
>>> ret = subprocess.call('date -u', shell=True)
Thu Jul 31 09:19:50 PM UTC 2025
```

You need that `shell=True` to recognize the command line `date -u`, splitting it into separate strings and possibly expanding any wildcard characters such as `*` (we didn't use any in this example).

The second method makes a list of the arguments, so it doesn't need to call the shell:

```
>>> ret = subprocess.call(['date', '-u'])
Thu Jul 31 09:20:19 PM UTC 2025
```

Create a Process with multiprocessing

You can run a Python function as a separate process, or even create multiple independent processes with the `multiprocessing` module. The sample code in Example 21-1 is short and simple; save it as *mp.py* and then run it by typing **`python mp.py`**.

Example 21-1. mp.py

```
import multiprocessing
import os

def whoami(what):
    print("Process %s says: %s" % (os.getpid(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
```

```
        p = multiprocessing.Process(target=whoami,
          args=("I'm function %s" % n,))
        p.start()
```

When I run this, my output looks like this:

```
Process 6224 says: I'm the main program
Process 6225 says: I'm function 0
Process 6226 says: I'm function 1
Process 6227 says: I'm function 2
Process 6228 says: I'm function 3
```

The `Process()` function spawned a new process and ran the `do_this()` function in it. Because we did this in a loop that had four passes, we generated four new processes that executed `do_this()` and then exited.

The `multiprocessing` module has more bells and whistles than a clown on a calliope. It's really intended for those times when you need to farm out a task to multiple processes to save overall time—for example, downloading web pages for scraping or resizing images. The module includes ways to queue tasks, enable intercommunication among processes, and wait for all the processes to finish.

Kill a Process with terminate()

If you created one or more processes and want to terminate one for some reason (perhaps it's stuck in a loop, or maybe you're bored, or you want to be an evil overlord), use `terminate()`. In Example 21-2, our process would count to a million, sleeping at each step for a second, and printing an irritating message. However, our main program runs out of patience in five seconds and nukes it from orbit.

Example 21-2. mp2.py

```
import multiprocessing
import time
import os

def whoami(name):
    print("I'm %s, in process %s" % (name, os.getpid()))

def loopy(name):
    whoami(name)
    start = 1
    stop = 1000000
    for num in range(start, stop):
        print("\tNumber %s of %s. Honk!" % (num, stop))
        time.sleep(1)

if __name__ == "__main__":
    whoami("main")
```

```
p = multiprocessing.Process(target=loopy, args=("loopy",))
p.start()
time.sleep(5)
p.terminate()
```

When I run this program, I get the following:

```
I'm main, in process 97080
I'm loopy, in process 97081
    Number 1 of 1000000. Honk!
    Number 2 of 1000000. Honk!
    Number 3 of 1000000. Honk!
    Number 4 of 1000000. Honk!
    Number 5 of 1000000. Honk!
```

Get System Info with os

The standard os package provides a lot of details on your system, and lets you control some of it if you run your Python script as a privileged user (root or administrator). Besides file and directory functions that are covered in Chapter 20, os has informational functions like these (run on a Chromebook):

```
>>> import os
>>> os.uname()
posix.uname_result(sysname='Linux',
nodename='penguin',
release='6.6.76-08096-g300882a0a131',
version='#1 SMP PREEMPT_DYNAMIC Sat, 24 May 2025 01:23:41 -0700',
machine='x86_64')
>>> os.getloadavg()
(0.0, 0.0, 0.1640625)
>>> os.cpu_count()
8
```

A useful function is `system()`, which executes a command string as though you typed it at a terminal:

```
>>> import os
>>> os.system('date -u')
Thu Jul 31 09:24:51 PM UTC 2025
0
```

It's a grab bag. See the docs for interesting tidbits.

Get Process Info with psutil

The third-party package psutil also provides system and process information for Linux, Unix, macOS, and Windows systems.

You can guess how to install it:

```
$ pip install psutil
```

Coverage includes the following:

System
: CPU, memory, disk, network, sensors

Processes
: ID, parent ID, CPU, memory, open files, threads

We already saw (in the previous `os` discussion) that my Chromebook has eight CPUs. I also ran that test on an iMac, which had four CPUs. How much time (in seconds) had they been using?

```
>>> import psutil
>>> psutil.cpu_times(True)
[scputimes(user=62306.49, nice=0.0, system=19872.71, idle=256097.64),
scputimes(user=19928.3, nice=0.0, system=6934.29, idle=311407.28),
scputimes(user=57311.41, nice=0.0, system=15472.99, idle=265485.56),
scputimes(user=14399.49, nice=0.0, system=4848.84, idle=319017.87)]
```

And how busy are they now?

```
>>> import psutil
>>> psutil.cpu_percent(True)
26.1
>>> psutil.cpu_percent(percpu=True)
[39.7, 16.2, 50.5, 6.0]
```

You might never need this kind of data, but it's good to know where to look if you do.

Command Automation

You often run commands from the shell (either manually typed commands or shell scripts), but Python has more than one good third-party management tool.

A related topic, *task queues*, is discussed in "Queues" on page 357.

Invoke

Version 1 of the `fabric` tool lets you define local and remote (networked) tasks with Python code. The developers split this original package into `fabric2` (remote) and `invoke` (local).

Install `invoke` by running the following:

```
$ pip install invoke
```

One use of `invoke` is to make functions available as command-line arguments. Let's make a *tasks.py* file with the lines shown in Example 21-3.

Example 21-3. tasks.py

```
from invoke import task

@task
def mytime(ctx):
    import time
    now = time.time()
    time_str = time.asctime(time.localtime(now))
    print("Local time is", time_str)
```

(That `ctx` argument is the first argument for each task function, but it's used only internally by `invoke`. It doesn't matter what you call it, but an argument needs to be there.)

Here's how it works:

```
$ invoke mytime
Local time is Sun Feb 16 21:32:50 2025
```

Use the argument `-l` or `--list` to see which tasks are available:

```
$ invoke -l
Available tasks:

  mytime
```

Tasks can have arguments, and you can invoke multiple tasks at one time from the command line (similar to `&&` use in shell scripts).

Other uses include the following:

- Running local shell commands with the `run()` function
- Responding to string output patterns of programs

This was a brief glimpse. See the docs for all the details.

Other Command Helpers

These Python packages have some similarity to `invoke`, but one or another may be a better fit when you need it:

- Click
- Doit
- sh
- Delegator.py
- Pypeln

Concurrency

The official Python site discusses concurrency in general and in the standard library. Those pages have many links to various packages and techniques.

In computers, if you're waiting for something, it's usually for one of two reasons:

I/O bound
: This reason is by far the most common. Computer CPUs are ridiculously fast—hundreds of times faster than computer memory and many thousands of times faster than disks or networks.

CPU bound
: The CPU keeps busy. This happens with *number-crunching* tasks such as scientific or graphic calculations.

Two more terms are related to concurrency:

Synchronous
: One task follows the other, like a line of goslings behind their parents.

Asynchronous
: Tasks are independent, like random geese splashing down in a pond.

As you progress from simple systems and tasks to real-life problems, you'll need at some point to deal with concurrency. Consider a website, for example. You can usually provide static and dynamic pages to web clients fairly quickly. A fraction of a second is considered interactive, but if the display or interaction takes longer, people become impatient. Tests by companies such as Google and Amazon showed that traffic drops off quickly if the page loads even a fraction of a second slower.

But what if you can't help it when something takes a long time, such as uploading a file, resizing an image, or querying a database? You can't perform those tasks within your synchronous web server code anymore, because someone's waiting.

On a single machine, if you want to perform multiple tasks as fast as possible, you should make them independent. Slow tasks shouldn't block all the others.

This chapter showed earlier how multiprocessing can be used to overlap work on a single machine. If you needed to resize an image, your web server code could call a separate, dedicated image-resizing process to run asynchronously and concurrently. It could scale your application horizontally by invoking multiple resizing processes.

The trick is getting them all to work with one another. Any shared control or state means that bottlenecks will occur. An even bigger trick is dealing with failures, because concurrent computing is harder than regular computing. Many more steps can go wrong, and your odds of end-to-end success are lower.

All right. Which techniques can help you deal with these complexities? Let's begin with a good way to manage multiple tasks: queues.

Queues

A *queue* is like a list: elements are added at one end and taken away from the other. The most common is referred to as *FIFO* (first in, first out).

Suppose that you're washing dishes. If you're stuck with the entire job, you need to wash each dish, dry it, and put it away. You can do this in various ways. You might wash the first dish, dry it, and then put it away. You then repeat with the second dish, and so on. Or, you might *batch* operations and wash all the dishes, dry them all, and then put them away; this assumes you have space in your sink and drainer for all the dishes that accumulate at each step. These are all synchronous approaches—one worker, one thing at a time.

As an alternative, you could get a helper or two. If you're the washer, you can hand each cleaned dish to the dryer, who hands each dried dish to the put-away-er. As long as each of you works at the same pace, you should finish much faster than by yourself.

However, what if you wash faster than the dryer dries? Wet dishes either fall on the floor, or you pile them up between you and the dryer, or you just whistle off-key until the dryer is ready. And if the last person is slower than the dryer, dry dishes can end up falling on the floor, or piling up, or the dryer does the whistling. You have multiple workers, but the overall task is still synchronous and can proceed only as fast as the slowest worker.

Many hands make light work, goes the old saying (it always makes me think of the Amish, who work as a community to raise/build a barn). Adding workers can build a barn or do the dishes, faster. This involves *queues*.

In general, queues transport *messages*, which can be any kind of information. In this case, we're interested in queues for distributed task management, also known as *work queues*, *job queues*, or *task queues*. Each dish in the sink is given to an available washer, who washes and hands it off to the first available dryer, who dries and hands it to a put-away-er. This can be synchronous (workers wait for a dish to handle and another worker to whom to give it), or asynchronous (dishes are stacked between workers with different paces). As long as you have enough workers, and they keep up with the dishes, the whole process moves a lot faster.

Processes

You can implement queues in many ways. For a single machine, the standard library's `multiprocessing` module (which you saw earlier) contains a `Queue` function. Let's simulate just a single washer and multiple dryer processes (someone can put the

dishes away later) and an intermediate `dish_queue`. Call this program *dishes.py* (Example 21-4).

Example 21-4. dishes.py

```
import multiprocessing as mp

def washer(dishes, output):
    for dish in dishes:
        print('Washing', dish, 'dish')
        output.put(dish)

def dryer(input):
    while True:
        dish = input.get()
        print('Drying', dish, 'dish')
        input.task_done()

dish_queue = mp.JoinableQueue()
dryer_proc = mp.Process(target=dryer, args=(dish_queue,))
dryer_proc.daemon = True
dryer_proc.start()

dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

Run your new program thusly:

```
$ python dishes.py
Washing salad dish
Washing bread dish
Washing entree dish
Washing dessert dish
Drying salad dish
Drying bread dish
Drying entree dish
Drying dessert dish
```

This queue looks a lot like a simple Python iterator, producing a series of dishes. It actually starts up separate processes along with the communication between the washer and dryer. I used a `JoinableQueue` and the final `join()` method to let the washer know that all the dishes have been dried. The `multiprocessing` module has other queue types, and you can read the documentation for more examples.

Threads

A *thread* runs within a process with access to everything in the process, similar to a multiple personality. The `multiprocessing` module has a cousin called `threading` that uses threads instead of processes (actually, `multiprocessing` was designed later as its process-based counterpart). Let's redo our process example with threads, as shown in Example 21-5.

Example 21-5. thread1.py

```
import threading

def do_this(what):
    whoami(what)

def whoami(what):
    print("Thread %s says: %s" % (threading.current_thread(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = threading.Thread(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

Here's what prints for me:

```
Thread <_MainThread(MainThread, started 140735207346960)> says: I'm the main
program
Thread <Thread(Thread-1, started 4326629376)> says: I'm function 0
Thread <Thread(Thread-2, started 4342157312)> says: I'm function 1
Thread <Thread(Thread-3, started 4347412480)> says: I'm function 2
Thread <Thread(Thread-4, started 4342157312)> says: I'm function 3
```

We can reproduce our process-based dish example by using threads, as shown in Example 21-6.

Example 21-6. thread_dishes.py

```
import threading, queue
import time

def washer(dishes, dish_queue):
    for dish in dishes:
        print ("Washing", dish)
        time.sleep(5)
        dish_queue.put(dish)

def dryer(dish_queue):
    while True:
```

```
        dish = dish_queue.get()
        print ("Drying", dish)
        time.sleep(10)
        dish_queue.task_done()


dish_queue = queue.Queue()
for n in range(2):
    dryer_thread = threading.Thread(target=dryer, args=(dish_queue,))
    dryer_thread.start()

dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

One difference between `multiprocessing` and `threading` is that `threading` does not have a `terminate()` function. There's no easy way to terminate a running thread, because it can cause all sorts of problems in your code, and possibly in the space-time continuum itself.

Threads can be dangerous. Like manual memory management in languages such as C and C++, they can cause bugs that are extremely hard to find, let alone fix. To use threads, all the code in the program (and in external libraries that it uses) must be *thread safe*. In the preceding example code, the threads didn't share any global variables, so they could run independently without breaking anything.

Imagine that you're a paranormal investigator in a haunted house. Ghosts roam the halls, but none are aware of the others, and at any time, any of them can view, add, remove, or move any of the house's contents.

You're walking apprehensively through the house, taking readings with your impressive instruments. Suddenly you notice that the candlestick you passed seconds ago is now missing.

The contents of the house are like the variables in a program. The ghosts are threads in a process (the house). If the ghosts only cast spectral glances at the house's contents, there would be no problem. It's like a thread reading the value of a constant or variable without trying to change it.

Yet, some unseen entity could grab your flashlight, blow cold air down your neck, put marbles on the stairs, or make the fireplace come ablaze. The *really* subtle ghosts would change things in other rooms that you might never notice.

Despite your fancy instruments, you'd have a very hard time figuring out who did what, how, when, and where.

If you used multiple processes instead of threads, it would be like having multiple houses but with only one (living) person in each. If you put your brandy in front of

the fireplace, it would still be there an hour later—some lost to evaporation or wobbling mice, but in the same place.

Threads can be useful and safe when global data is not involved. In particular, threads are useful for saving time while waiting for an I/O operation to complete. In these cases, they don't have to fight over data, because each has completely separate variables.

But threads do sometimes have good reasons to change global data. In fact, one common reason to launch multiple threads is to let them divide up the work on some data, so a certain degree of change to the data is expected.

The usual way to share data safely is to apply a software *lock* before modifying a variable in a thread. This keeps the other threads out while the change is made. It's like having a Ghostbuster guard the room you want to remain unhaunted. The trick, though, is that you need to remember to release the lock. Plus, locks can be nested: what if another Ghostbuster is also watching the same room or the house itself? The use of locks is traditional but hard to get right.

The GIL

Python has a controversial feature called the *Global Interpreter Lock* (*GIL*). This was, essentially, an early hack added to avoid threading problems in the Python interpreter. However, it can make a multithreaded program slower than its single-threaded counterpart or even a multiprocess version. Even worse, the GIL limits the interpreter to running on a single core CPU, and modern computers all have multiple cores.

If you're running C code in an extension, you're safe: C code doesn't modify Python code running in the interpreter. This is why C extensions like NumPy and pandas can run faster than native Python code. Chapter 27 discusses these, and other intriguing possibilities, like the new "Python++" language called Mojo.

Many people have tried to kill the GIL, but all the attempts slowed single-threaded code, which is the most common case. A solution has finally been agreed upon but will be introduced gradually: "PEP 703–Making the Global Interpreter Lock Optional in CPython". Python 3.13 introduces an experimental GIL-less Python version. Depending on how this works out with many developers banging on it, version 3.14 is expected to go further.

So for now in Python, the recommendations are as follows:

- Use threads for I/O-bound problems.
- Use processes, networking, or events (discussed in the next section) for CPU-bound problems.

Concurrent.futures

As you've just seen, using threads or multiple processes involves multiple details. The `concurrent.futures` module was added to the Python 3.2 standard library to simplify these. It lets you schedule an asynchronous pool of workers, using threads (when I/O-bound) or processes (when CPU-bound). You get back a *future* to track their state and collect the results.

Example 21-7 contains a test program that you can save as *cf.py*. The task function `calc()` sleeps for one second (our way of faking being busy with something), calculates the square root of its argument, and returns it. The program takes an optional command-line argument of the number of workers to use, which defaults to 3. The program starts this number of workers in a thread pool, then a process pool, and then prints the elapsed times. The `values` list contains five numbers, sent to `calc()` one at a time in a worker thread or process.

Example 21-7. cf.py

```
from concurrent import futures
import math
import time
import sys

def calc(val):
    time.sleep(1)
    result = math.sqrt(float(val))
    return result

def use_threads(num, values):
    t1 = time.time()
    with futures.ThreadPoolExecutor(num) as tex:
        results = tex.map(calc, values)
    t2 = time.time()
    return t2 - t1

def use_processes(num, values):
    t1 = time.time()
    with futures.ProcessPoolExecutor(num) as pex:
        results = pex.map(calc, values)
    t2 = time.time()
    return t2 - t1

def main(workers, values):
    print(f"Using {workers} workers for {len(values)} values")
    t_sec = use_threads(workers, values)
    print(f"Threads took {t_sec:.4f} seconds")
    p_sec = use_processes(workers, values)
    print(f"Processes took {p_sec:.4f} seconds")
```

```
if __name__ == '__main__':
    workers = int(sys.argv[1])
    values = list(range(1, 6)) # 1 .. 5
    main(workers, values)
```

Here are some results that I got:

```
$ python cf.py 1
Using 1 workers for 5 values
Threads took 5.0736 seconds
Processes took 5.5395 seconds
$ python cf.py 3
Using 3 workers for 5 values
Threads took 2.0040 seconds
Processes took 2.0351 seconds
$ python cf.py 5
Using 5 workers for 5 values
Threads took 1.0052 seconds
Processes took 1.0444 seconds
```

That one-second `sleep()` forced each worker to take a second for each calculation:

- With only one worker at a time, everything was serial, and the total time was more than five seconds.
- Five workers matched the size of the values being tested, so we had an elapsed time of just more than a second.
- With three workers, we needed two runs to handle all five values, so two seconds elapsed.

In the program, I ignore the actual `results` (the square roots that we calculated) to emphasize the elapsed times. Also, using `map()` to define the pool causes us to wait for all workers to finish before returning `results`. To get each result as it is completed, let's try another test (call it *cf2.py*) in which each worker returns the value and its square root as soon as it calculates it (Example 21-8).

Example 21-8. cf2.py

```
from concurrent import futures
import math
import sys

def calc(val):
    result = math.sqrt(float(val))
    return val, result

def use_threads(num, values):
    with futures.ThreadPoolExecutor(num) as tex:
        tasks = [tex.submit(calc, value) for value in values]
```

```
        for f in futures.as_completed(tasks):
            yield f.result()

def use_processes(num, values):
    with futures.ProcessPoolExecutor(num) as pex:
        tasks = [pex.submit(calc, value) for value in values]
        for f in futures.as_completed(tasks):
            yield f.result()

def main(workers, values):
    print(f"Using {workers} workers for {len(values)} values")
    print("Using threads:")
    for val, result in use_threads(workers, values):
        print(f'{val} {result:.4f}')
    print("Using processes:")
    for val, result in use_processes(workers, values):
        print(f'{val} {result:.4f}')

if __name__ == '__main__':
    workers = 3
    if len(sys.argv) > 1:
        workers = int(sys.argv[1])
    values = list(range(1, 6)) # 1 .. 5
    main(workers, values)
```

Our `use_threads()` and `use_processes()` functions are now generator functions that call `yield` to return on each iteration. From one run on my machine, you can see how the workers don't always finish `1` through `5` in order:

```
$ python cf2.py 5
Using 5 workers for 5 values
Using threads:
3 1.7321
1 1.0000
2 1.4142
4 2.0000
5 2.2361
Using processes:
1 1.0000
2 1.4142
3 1.7321
4 2.0000
5 2.2361
```

You can use `concurrent.futures` anytime you want to launch a bunch of concurrent tasks, such as the following:

- Crawling URLs on the web
- Processing files, such as resizing images
- Calling service APIs

As usual, the docs provide additional details but are much more technical.

Green Threads and gevent

As you've seen, developers traditionally avoid slow spots in programs by running them in separate threads or processes. The Apache web server is an example of this design.

One alternative is event-based programming. An *event-based* program runs a central *event loop*, doles out any tasks, and repeats the loop. The NGINX web server follows this design and is generally faster than Apache.

The gevent library is event based and accomplishes a neat trick: you write normal imperative code, and it magically converts pieces to *coroutines*. These are like generators that can communicate with one another and keep track of where they are. The gevent library modifies many of Python's standard objects such as `socket` to use its mechanism instead of blocking. This does not work with Python add-in code that was written in C, as some database drivers are.

You install gevent by using `pip`:

```
$ pip install gevent
```

Example 21-9 is a variation of sample code at the gevent website. You'll see the `socket` module's `gethostbyname()` function in the upcoming DNS section. This function is synchronous, so you wait (possibly many seconds) while it chases name servers around the world to look up that address. But you could use the gevent version to look up multiple sites independently. Save this as *gevent_test.py*.

Example 21-9. gevent_test.py

```
import gevent
from gevent import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(gevent.socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

A one-line `for` loop is in the preceding example. Each hostname is submitted in turn to a `gethostbyname()` call, but each can run asynchronously because it's the gevent version of `gethostbyname()`.

Run *gevent_test.py*:

```
$ python gevent_test.py
66.6.44.4
74.125.142.121
78.136.12.50
```

The `gevent.spawn()` method creates a *greenlet* (also known as a *green thread* or a *microthread*) to execute each `gevent.socket.gethostbyname(url)`.

The difference from a normal thread is that a greenlet doesn't block. If something occurred that would have blocked a normal thread, gevent switches control to one of the other greenlets.

The `gevent.joinall()` method waits for all the spawned jobs to finish. Finally, we dump the IP addresses that we got for these hostnames.

Instead of the gevent version of `socket`, you can use its evocatively named *monkey-patching* functions. These modify standard modules such as `socket` to use greenlets rather than calling the gevent version of the module. This is useful when you want gevent to be applied all the way down, even into code that you might not be able to access.

At the top of your program, add the following call:

```
from gevent import monkey
monkey.patch_socket()
```

This inserts the gevent socket everywhere the normal `socket` is called, anywhere in your program, even in the standard library. Again, this works only for Python code, not libraries written in C.

Another function monkey-patches even more standard library modules:

```
from gevent import monkey
monkey.patch_all()
```

Use this at the top of your program to get as many gevent speedups as possible.

Save Example 21-10 as *gevent_monkey.py*.

Example 21-10. gevent_monkey.py

```
import gevent
from gevent import monkey; monkey.patch_all()
import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

Again, run the program:

```
$ python gevent_monkey.py
66.6.44.4
74.125.192.121
78.136.12.50
```

Using gevent has potential dangers. As with any event-based system, each chunk of code that you execute should be relatively quick. Although it's nonblocking, code that does a lot of work is still slow.

The very idea of monkey-patching makes some people nervous. Yet, many large sites such as Pinterest use gevent to speed up their sites significantly. Like the fine print on a bottle of pills, use gevent as directed.

For more examples, see the thorough "gevent for the Working Python Developer" tutorial.

You might also want to consider Tornado or Gunicorn, two other popular event-driven frameworks. They provide both the low-level event handling and a fast web server. They're worth a look if you'd like to build a fast website without messing with a traditional web server such as Apache.

Twisted

Twisted is an asynchronous, event-driven networking framework. You connect functions to events such as data received or connection closed, and those functions are called when those events occur.

This is a *callback* design, and if you've written anything in JavaScript, it might seem familiar. If it's new to you, it can seem backward. For some developers, callback-based code becomes harder to manage as the application grows.

You install Twisted by running the following:

```
$ pip install twisted
```

Twisted is a large package, with support for many internet protocols on top of TCP and UDP. To be short and simple, we show a little knock-knock server and client, adapted from Twisted examples. First, let's look at the server, *knock_server.py* (Example 21-11).

Example 21-11. knock_server.py

```
from twisted.internet import protocol, reactor

class Knock(protocol.Protocol):
    def dataReceived(self, data):
        print('Client:', data)
        if data.startswith("Knock knock"):
            response = "Who's there?"
        else:
            response = data + " who?"
        print('Server:', response)
        self.transport.write(response)

class KnockFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Knock()

reactor.listenTCP(8000, KnockFactory())
reactor.run()
```

Now let's take a glance at its trusty companion, *knock_client.py* (Example 21-12).

Example 21-12. knock_client.py

```
from twisted.internet import reactor, protocol

class KnockClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("Knock knock")

    def dataReceived(self, data):
        if data.startswith("Who's there?"):
            response = "Disappearing client"
            self.transport.write(response)
        else:
            self.transport.loseConnection()
            reactor.stop()

class KnockFactory(protocol.ClientFactory):
    protocol = KnockClient

def main():
    f = KnockFactory()
    reactor.connectTCP("localhost", 8000, f)
    reactor.run()

if __name__ == '__main__':
    main()
```

Start the server first:

```
$ python knock_server.py
```

Then start the client:

```
$ python knock_client.py
```

The server and client exchange messages, and the server prints the conversation:

```
Client: Knock knock
Server: Who's there?
Client: Disappearing client
Server: Disappearing client who?
```

Our trickster client then ends, keeping the server waiting for the punch line.

If you'd like to enter the Twisted passages, try some of the other examples from its documentation.

asyncio

Python added the asyncio library in version 3.4. It's a way of defining concurrent code by using the new `async` and `await` capabilities.

Like most programming languages, Python has been *synchronous*. It runs through code linearly, a line at a time, from top to bottom. When you call a function, Python jumps into its code, and the caller waits until the function returns before resuming what it was doing.

Your CPU can do only one thing at a time, so synchronous execution makes perfect sense. But it turns out that often a program is not actually running any code, but waiting for something, like data from a file or a network service. This is like us staring at a browser screen while waiting for a site to load. If we could avoid this "busy waiting," we might shorten the total time of our programs. This is also called improving *throughput*.

Your concurrency choices have included threads, processes, or a third-party solution like gevent or Twisted. But now a growing number of *asynchronous* answers have been built in to Python and third-party solutions. These coexist with the usual synchronous Python code, but, to borrow a *Ghostbusters* warning, you can't cross the streams. I'll show you how to avoid any ectoplasmic side effects.

Coroutines and Event Loops

In Python 3.4, Python added the standard *asynchronous* module `asyncio`. Python 3.5 then added the keywords `async` and `await`. These implement some new concepts:

Coroutines
: Functions that pause at various points

An event loop
: Code that schedules and runs coroutines

These let us write asynchronous code that looks something like the normal synchronous code that we're used to. Otherwise, we'd need to use one of the approaches mentioned in Chapter 22.

Normal multitasking is what your operating system does to your processes. It decides what's fair, who's being a CPU hog, when to open the I/O spigots, and so on. The event loop, however, provides *cooperative multitasking*, in which coroutines indicate when they're able to start and stop. They run in a single thread, so you don't have the potential issues that I mentioned in "Threads" on page 359.

You *define* a coroutine by putting `async` before its initial `def`. You *call* a coroutine by doing one of the following:

- Putting `await` before it, which quietly adds the coroutine to an existing event loop. You can do this only within another coroutine.
- Using `asyncio.run()`, which explicitly starts an event loop.
- Using `asyncio.create_task()` or `asyncio.ensure_future()`.

This example uses the first two calling methods:

```
>>> import asyncio
>>>
>>> async def wicked():
...     print("Surrender,")
...     await asyncio.sleep(2)
...     print("Dorothy!")
...
>>> asyncio.run(wicked())
Surrender,
Dorothy!
```

A dramatic two-second wait occurred in there that you can't see on a printed page. Let's prove that we didn't cheat (see Chapter 27 for `timeit` details):

```
>>> from timeit import timeit
>>> timeit("asyncio.run(wicked())", globals=globals(), number=1)
Surrender,
Dorothy!
2.005701574998966
```

That `asyncio.sleep(2)` call was itself a coroutine, just an example here to fake something time-consuming like an API call.

The line `asyncio.run(wicked())` is a way of running a coroutine from synchronous Python code (here, the top level of the program).

The difference from a standard synchronous counterpart (using `time.sleep()`) is that the caller of `wicked()` is not blocked for two seconds while it runs.

The third way to run a coroutine is to create a *task* and `await` it. This example shows the task approach along with the previous two methods:

```
>>> import asyncio
>>>
>>> async def say(phrase, seconds):
...     print(phrase)
...     await asyncio.sleep(seconds)
...
>>> async def wicked():
...     task_1 = asyncio.create_task(say("Surrender,", 2))
...     task_2 = asyncio.create_task(say("Dorothy!", 0))
...     await task_1
...     await task_2
...
>>> asyncio.run(wicked())
Surrender,
Dorothy!
```

If you run this, you'll see that no delay occurs between the two lines printing this time. That's because they are separate tasks; `task_1` pauses two seconds after printing `Surrender`, but that doesn't affect `task_2`.

An `await` is similar to a `yield` in a generator, but rather than returning a value, it marks a spot where the event loop can pause it if needed.

There's lots more where this came from in the docs. Synchronous and asynchronous code can coexist in the same program. Just remember to put `async` before the `def`, and `await` before the call of your asynchronous function.

Here's some more information:

- A list of many `asyncio` links, compiled by Dylan Hogg
- Code for an `asyncio` web crawler, by A. Jesse Jiryu Davis and Guido van Rossum

Asyncio Alternatives

Although asyncio is a standard Python package, you can use `async` and `await` without it. Coroutines and the event loop are independent. The design of asyncio is sometimes criticized, and third-party alternatives have appeared:

- Curio
- Trio

Example 21-13 is a real example using Trio and asks (an async web framework, modeled on the `requests` API). This is a concurrent web-crawling example using Trio and asks, adapted from a Stack Overflow answer. To run this, first `pip install` both Trio and asks.

Example 21-13. trio_asks_sites.py

```
import time

import asks
import trio

asks.init("trio")

urls = [
    'https://boredomtherapy.com/bad-taxidermy/',
    'http://www.badtaxidermy.com/',
    'https://crappytaxidermy.com/',
    'https://www.ranker.com/list/bad-taxidermy-pictures/ashley-reign',
]

async def get_one(url, t1):
    r = await asks.get(url)
    t2 = time.time()
    print(f"{(t2-t1):.04}\t{len(r.content)}\t{url}")

async def get_sites(sites):
    t1 = time.time()
    async with trio.open_nursery() as nursery:
        for url in sites:
            nursery.start_soon(get_one, url, t1)

if __name__ == "__main__":
```

```
print("seconds\tbytes\turl")
trio.run(get_sites, urls)
```

Here's what I got:

```
$ python trio_asks_sites.py
seconds bytes   url
0.1287  5735    https://boredomtherapy.com/bad-taxidermy/
0.2134  146082  https://www.ranker.com/list/bad-taxidermy-pictures/ashley-reign
0.215   11029   http://www.badtaxidermy.com/
0.3813  52385   https://crappytaxidermy.com/
```

You'll notice that Trio did not use `asyncio.run()`, but instead its own `trio.open_nursery()`. If you're curious, you can read "Some thoughts on asynchronous API design in a post-async/await world" by Nathaniel J. Smith and the Stack Overflow discussion "What is the core difference between asyncio and trio?" about the design decisions behind Trio.

A new package called AnyIO provides a single interface to asyncio, Curio, and Trio.

In the future, you can expect more async approaches, both in standard Python and from third-party developers.

Async Versus...

As you've seen in many places in this book, many techniques can be used for concurrency. How does the async stuff compare with them?

Processes
: This is a good solution if you want to use all the CPU cores on your machine or multiple machines. But processes are heavy, take a while to start, and require serialization for interprocess communication.

Threads
: Although threads were designed as a lightweight alternative to processes, each thread uses a good chunk of memory. They are also thwarted by the GIL.

Coroutines
: Coroutines are much lighter than threads; you can create hundreds of thousands of coroutines on a machine that might support only a few thousand threads.

Green threads
: Green threads like gevent work well and look like synchronous code, but they require *monkey-patching* standard Python functions, such as socket libraries.

Callbacks
: Libraries like Twisted rely on *callbacks*: functions that are called when certain events occur. This is familiar to GUI and JavaScript programmers. The "inside-out" design of such systems can get messy.

Queues
: These tend to be a large-scale solution, when your data or processes really need more than one machine.

Mystery guest
: A dark horse like the new Mojo language may cut the Gordian concurrency knot. (See Chapter 27; Mojo looks like Python but is compiled, faster, and GIL-less.)

And the winner is … I don't know! I admit that I'm tempted by Mojo, but I'll quote that futurist baseball player, manager, and coach:

> It's tough to make predictions, especially about the future.
>
> —Yogi Berra

Async Frameworks and Servers

The async additions to Python are recent, and it's taking time for developers to create async versions of frameworks like Flask.

The Asynchronous Server Gateway Interface (ASGI) standard is an async version of WSGI, discussed further in "Hello, ASGI" by encode.io.

Here are some ASGI web servers:

- Hypercorn
- Sanic
- Uvicorn

And some async web frameworks:

- aiohttp—Client *and* server
- API-Hour
- asks—Like Requests
- BlackSheep
- Channels
- FastAPI—Uses type annotations
- Quart
- Responder

- Sanic
- Starlette
- Tornado
- Vibora

Finally, here are some async database interfaces:

- aiomysql
- aioredis
- asyncpg

Redis

Our earlier dishwashing code examples, using processes or threads, were run on a single machine. Let's take another approach to queues that can run on a single machine or across a network. Even with multiple singing processes and dancing threads, sometimes one machine isn't enough. You can treat this section as a bridge between single-box (one machine) and multiple-box concurrency.

To try the examples in this section, you'll need a Redis server and its Python module. You can see where to get them in Chapter 23. In that chapter, Redis's role is that of a database. Here, I'm featuring its concurrency personality.

A quick way to make a queue is with a Redis list. A Redis server runs on one machine; this can be the same one as its clients or another that the clients can access through a network. In either case, clients talk to the server via TCP, so they're networking. One or more provider clients pushes messages onto one end of the list. One or more client workers watches this list with a *blocking pop* operation. If the list is empty, they all just sit around playing cards. As soon as a message arrives, the first eager worker gets it.

Like our earlier process- and thread-based examples, *redis_washer.py* generates a sequence of dishes (Example 21-14).

Example 21-14. redis_washer.py

```
import redis
conn = redis.Redis()
print('Washer is starting')
dishes = ['salad', 'bread', 'entree', 'dessert']
for dish in dishes:
    msg = dish.encode('utf-8')
    conn.rpush('dishes', msg)
    print('Washed', dish)
```

```
conn.rpush('dishes', 'quit')
print('Washer is done')
```

The loop generates four messages containing a dish name, followed by a final message that says "quit." The loop appends each message to a list called `dishes` in the Redis server, similar to appending to a Python list.

And as soon as the first dish is ready, *redis_dryer.py* does its work (Example 21-15).

Example 21-15. redis_dryer.py

```
import redis
conn = redis.Redis()
print('Dryer is starting')
while True:
    msg = conn.blpop('dishes')
    if not msg:
        break
    val = msg[1].decode('utf-8')
    if val == 'quit':
        break
    print('Dried', val)
print('Dishes are dried')
```

This code waits for messages whose first token is "dishes" and prints that each one is dried. The code then obeys the `quit` message by ending the loop.

Next, we start the dryer and then the washer. Using the `&` at the end puts the first program in the *background*; it keeps running but doesn't listen to the keyboard anymore. This works on Linux, macOS, and Windows, although you might see different output on the next line. In this case (macOS), the output shows some information about the background dryer process. Then we start the washer process normally (in the *foreground*). You'll see the mingled output of the two processes:

```
$ python redis_dryer.py &
[2] 81691
Dryer is starting
$ python redis_washer.py
Washer is starting
Washed salad
Dried salad
Washed bread
Dried bread
Washed entree
Dried entree
Washed dessert
Washer is done
Dried dessert
Dishes are dried
[2]+  Done                    python redis_dryer.py
```

As soon as dish IDs start arriving at Redis from the washer process, our hard-working dryer process starts pulling them back out. Each dish ID is a number, except the final sentinel value, the string `quit`. When the dryer process reads that `quit` dish ID, it quits, and more background process information prints to the terminal (also system dependent). You can use a *sentinel* (an otherwise invalid value) to indicate something special from the data stream itself—in this case, that we're done. Otherwise, we'd need to add a lot more program logic, such as the following:

- Agreeing ahead of time on a maximum dish number, which would kind of be a sentinel anyway
- Doing a special *out-of-band* (not in the data stream) interprocess communication
- Timing out after a certain interval with no new data

Let's make a few last changes:

- Create multiple `dryer` processes.
- Add a timeout to each dryer rather than looking for a sentinel.

Example 21-16 shows the new *redis_dryer2.py*.

Example 21-16. redis_dryer2.py

```
def dryer():
    import redis
    import os
    import time
    conn = redis.Redis()
    pid = os.getpid()
    timeout = 20
    print('Dryer process %s is starting' % pid)
    while True:
        msg = conn.blpop('dishes', timeout)
        if not msg:
            break
        val = msg[1].decode('utf-8')
        if val == 'quit':
            break
        print('%s: dried %s' % (pid, val))
        time.sleep(0.1)
    print('Dryer process %s is done' % pid)

import multiprocessing
DRYERS=3
for num in range(DRYERS):
    p = multiprocessing.Process(target=dryer)
    p.start()
```

Start the dryer processes in the background and then the washer process in the foreground:

```
$ python redis_dryer2.py &
Dryer process 44447 is starting
Dryer process 44448 is starting
Dryer process 44446 is starting
$ python redis_washer.py
Washer is starting
Washed salad
44447: dried salad
Washed bread
44448: dried bread
Washed entree
44446: dried entree
Washed dessert
Washer is done
44447: dried dessert
```

One dryer process reads the `quit` ID and quits:

```
Dryer process 44448 is done
```

After 20 seconds, the other dryer processes get a return value of `None` from their `blpop` calls, indicating that they've timed out. They say their last words and exit:

```
Dryer process 44447 is done
Dryer process 44446 is done
```

After the last dryer subprocess quits, the main dryer program ends:

```
[1]+  Done                    python redis_dryer2.py
```

Beyond Queues

With more moving parts, our lovely assembly lines have more chances to be disrupted. If we need to wash the dishes from a banquet, do we have enough workers? What if the dryers get drunk? What if the sink clogs? Worries, worries!

How will you cope with it all? Common techniques include these:

Fire and forget
: Just pass dishes on and don't worry about the consequences, even if no one is there. That's the dishes-on-the-floor approach.

Request-reply
: The washer receives an acknowledgment from the dryer, and the dryer from the put-away-er, for each dish in the pipeline.

Back pressure or throttling
: This technique directs a fast worker to take it easy if someone downstream can't keep up.

In real systems, you need to be careful that workers are keeping up with the demand; otherwise, you'll hear the dishes hitting the floor. You might add new tasks to a *pending* list, while a worker process pops the latest message and adds it to a *working* list. When the message is done, it's removed from the working list and added to a *completed* list. This lets you know which tasks have failed or are taking too long. You can do this with Redis yourself, or use a system that someone else has already written and tested. Python-based queue packages that add this extra level of management include the following:

- Celery can execute distributed tasks synchronously or asynchronously, using the approaches we've discussed: multiprocessing, gevent, and others.
- RQ is a Python library for job queues, also based on Redis.

Review/Preview

This was a tough chapter. Processes, system and green threads, async, … help! My brain filled up by the second page. Concurrency is one of the more complex areas of Python.

Next up: networks, from low-level TCP/IP to mighty services.

Practice

21.1 Use multiprocessing to create three separate processes. Make each one wait a random number of seconds from 0 to 1, print the current time, and then exit.

CHAPTER 22

Data in Space: Networks

> *Time is nature's way of keeping everything from happening at once.*
> *Space is what prevents everything from happening to me.*
>
> —Quotes About Time

In Chapter 21, you read about *concurrency*: how to do more than one thing at a time. Now we'll try to do things in more than one place: *distributed computing* or *networking*. There are many good reasons to challenge time and space:

Performance
: Your goal is to keep fast components busy, not wait for slow ones.

Robustness
: There's safety in numbers, so you want to duplicate tasks to work around hardware and software failures.

Simplicity
: It's best practice to break complex tasks into many little ones that are easier to create, understand, and fix.

Scalability
: Increase your servers to handle load; decrease them to save money.

In this chapter, we work our way up from networking primitives to higher-level concepts. Let's start with TCP/IP and sockets.

TCP/IP

The internet is based on rules about how to make connections, exchange data, terminate connections, handle timeouts, and so on. These are called *protocols*, and they are arranged in *layers*. The purpose of layers is to allow innovation and alternative ways

of doing things; you can do anything you want on one layer as long as you follow the conventions in dealing with the layers above and below you.

The very lowest layer governs aspects such as electrical signals; each higher layer builds on those below. In the middle, more or less, is the Internet Protocol (IP) layer, which specifies the way network locations are addressed and *packets* (chunks) of data flow. In the layer above that, two protocols describe how to move bytes between locations:

User Datagram Protocol (UDP)
: This is used for short exchanges. A *datagram* is a tiny message sent in a single burst, like a note on a postcard.

Transmission Control Protocol (TCP)
: This protocol is used for longer-lived connections. It sends *streams* of bytes and ensures that they arrive in order without duplication.

UDP messages are not acknowledged, so you're never sure whether they arrive at their destination. Say you want to tell a joke over UDP:

```
Here's a UDP joke. Get it?
```

TCP sets up a secret handshake between sender and receiver to ensure a good connection. A TCP joke would start like this:

```
Do you want to hear a TCP joke?
Yes, I want to hear a TCP joke.
Okay, I'll tell you a TCP joke.
Okay, I'll hear a TCP joke.
Okay, I'll send you a TCP joke now.
Okay, I'll receive the TCP joke now.
... (and so on)
```

Your local machine always has the IP address `127.0.0.1` and the name `localhost`. You might see this called the *loopback interface*. If it's connected to the internet, your machine will also have a *public* IP. If you're just using a home computer, it's behind equipment such as a cable modem or router. You can run internet protocols even between processes on the same machine.

Most of the internet with which we interact—the web, database servers, and so on—is based on TCP running atop IP; for brevity, TCP/IP. Let's first look at some basic internet services. After that, we explore general networking patterns.

Sockets

If you like to know how things work, all the way down, this section is for you.

The lowest level of network programming uses a *socket*, borrowed from the C language and the Unix operating system. Socket-level coding is tedious. You'll have more

fun using something like ZeroMQ, but it's useful to see what lies beneath. For instance, messages about sockets often turn up when networking errors take place.

Let's write a simple client-server exchange, once with UDP and once with TCP. In the UDP example, the client sends a string in a UDP datagram to a server, and the server returns a packet of data containing a string. The server needs to listen at a particular address and port—which are like a post office and a post office box. The client needs to know these two values to deliver its message and receive any reply.

In the following client and server code, `address` is a tuple of (*`address`*, *`port`*). The `address` is a string, which can be a name or an *IP address*. When your programs are just talking to one another on the same machine, you can use the name `localhost` or the equivalent address string `127.0.0.1`.

First, let's send a little data from one process to another and return a little data back to the originator. The first program is the server, and the second is the client. In each program, we print the time and open a socket. The server will listen for connections to its socket, and the client will write to its socket, which transmits a message to the server.

Example 22-1 presents the first program, *udp_server.py*.

Example 22-1. udp_server.py

```
from datetime import datetime
import socket

server_address = ('localhost', 6789)
max_size = 4096

print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(server_address)

data, client = server.recvfrom(max_size)

print('At', datetime.now(), client, 'said', data)
server.sendto(b'Are you talking to me?', client)
server.close()
```

The server has to set up networking through two methods imported from the socket package. The first method, `socket.socket`, creates a socket, and the second, `bind`, *binds* to it (listens to any data arriving at that IP address and port). `AF_INET` means we'll create an IP socket. (There's another type for *Unix domain sockets*, but those work only on the local machine.) `SOCK_DGRAM` means we'll send and receive datagrams—in other words, we'll use UDP.

At this point, the server sits and waits for a datagram to come in (`recvfrom`). When one arrives, the server wakes up and gets both the data and information about the client. The `client` variable contains the address and port combination needed to reach the client. The server ends by sending a reply and closing its connection.

Let's take a look at *udp_client.py* (Example 22-2).

Example 22-2. udp_client.py

```
import socket
from datetime import datetime

server_address = ('localhost', 6789)
max_size = 4096

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.sendto(b'Hey!', server_address)
data, server = client.recvfrom(max_size)
print('At', datetime.now(), server, 'said', data)
client.close()
```

The client has most of the same methods as the server (with the exception of `bind()`). The client sends and then receives, whereas the server receives first.

Start the server first, in its own window. The server will print its greeting and then wait with an eerie calm until a client sends it some data:

```
$ python udp_server.py
Starting the server at 2025-07-31 16:29:18.279874
Waiting for a client to call.
```

Next, start the client in another window. The client will print its greeting, send data (the bytes value `Hey`) to the server, print the reply, and then exit:

```
$ python udp_client.py
Starting the client at 2025-07-31 16:29:23.962248
At 2025-07-31 16:29:23.963004 ('127.0.0.1', 6789) said b'Are you talking to me?'
```

Finally, the server will print the message it received and exit:

```
At 2025-07-31 16:29:23.962813 ('127.0.0.1', 36905) said b'Hey!'
```

Why is there a `b` before the `Are you talking to me?` and the `Hey!`? Because they're bytes, not strings. These low-level functions send and receive binary data.

The client needs to know the server's address and port number but doesn't need to specify a port number for itself. That is automatically assigned by the system—in this case, the port number is `56267`.

UDP sends data in single chunks. It does *not* guarantee delivery. If you send multiple messages via UDP, they can arrive out of order or not at all. UDP is fast, light, connectionless, and unreliable (I've known some people like that). UDP is useful when you need to push packets quickly, and can tolerate a lost packet now and then, such as with voice over IP (VoIP).

Which brings us to TCP. TCP is used for longer-lived connections, such as the web. This protocol delivers data in the order in which you send it. If any problems occur, TCP tries to send the data again. This makes TCP a bit slower than UDP, but usually a better choice when you need all the packets in the right order.

The first two versions of the web protocol HTTP were based on TCP, but HTTP/3 is based on a protocol called QUIC, which itself uses UDP. So choosing between UDP and TCP can involve many factors.

Let's shoot a few packets from client to server and back with TCP.

tcp_client.py acts like the previous UDP client, sending only one string to the server, but there are small differences in the socket calls, illustrated in Example 22-3.

Example 22-3. tcp_client.py

```
import socket
from datetime import datetime

address = ('localhost', 6789)
max_size = 1000

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(address)
client.sendall(b'Hey!')
data = client.recv(max_size)
print('At', datetime.now(), 'someone replied', data)
client.close()
```

We've replaced `SOCK_DGRAM` with `SOCK_STREAM` to get the streaming protocol, TCP. We also added a `connect()` call to set up the stream. We didn't need that for UDP because each datagram was on its own in the wild, wooly Internet.

As Example 22-4 demonstrates, *tcp_server.py* also differs from its UDP cousin.

Example 22-4. tcp_server.py

```
from datetime import datetime
import socket

address = ('localhost', 6789)
max_size = 1000

print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(address)
server.listen(5)

client, addr = server.accept()
data = client.recv(max_size)

print('At', datetime.now(), client, 'said', data)
client.sendall(b'Are you talking to me?')
client.close()
server.close()
```

The `server.listen(5)` function is configured to queue up to five client connections before refusing new ones. The `server.accept()` function gets the first available message as it arrives.

The `client.recv(1000)` function sets a maximum acceptable message length of one thousand bytes.

As you did earlier, start the server and then the client, and watch the fun. First, the server:

```
$ python tcp_server.py
Starting the server at 2025-07-31 16:32:44.003119
Waiting for a client to call.
```

Now, start the client. It will send its message to the server, receive a response, and then exit:

```
$ python tcp_client.py
Starting the client at 2025-07-31 16:33:21.395920
At 2025-07-31 16:33:21.397052 someone replied b'Are you talking to me?'
```

The server collects the message, prints it, responds, and then quits:

```
At 2025-07-31 16:33:21.396797 <socket.socket fd=4, family=2, type=1,
proto=0, laddr=('127.0.0.1', 6789), raddr=('127.0.0.1', 45362)> said
b'Hey!'
```

Notice that the TCP server calls `client.sendall()` to respond, and the earlier UDP server calls `client.sendto()`. TCP maintains the client-server connection across multiple socket calls and remembers the client's IP address.

This doesn't look so bad, but if you try to write anything more complex, you'll see how sockets really operate at a low level:

- UDP sends messages, but their size is limited and they're not guaranteed to reach their destination.
- TCP sends streams of bytes, not messages. You don't know the number of bytes the system will send or receive with each call.
- To exchange entire messages with TCP, you need some extra information to reassemble the full message from its segments: a fixed message size (bytes), or the size of the full message, or a delimiting character.
- Because messages are bytes, not Unicode text strings, you need to use the Python `bytes` type.

After all this, if you find yourself interested in socket programming, check out the Python socket programming HOWTO by Gordon McMillan for more details.

Scapy

Sometimes you need to dip into the networking stream and watch the bytes swimming by. You may want to debug a web API or track down a security issue. The Scapy library and program provide a domain-specific language to create and inspect packets in Python, which is much easier than writing and debugging the equivalent C programs.

A standard installation uses `pip install scapy`. The docs are extremely thorough. If you use tools like tcpdump or Wireshark to investigate TCP issues, you should look at Scapy. Finally, don't confuse Scapy with Scrapy, which is covered in "Crawl and Scrape" on page 484.

Netcat

Another tool to test networks and ports is Netcat, often abbreviated to nc. Here's an example of an HTTP connection to Google's website, and requesting some basic information about its home page; I'm replacing most of the returned headers with an ellipsis (…):

```
$ nc www.google.com 80
HEAD / HTTP/1.1

HTTP/1.1 200 OK
...
Date: Mon, 17 Feb 2025 03:53:08 GMT
...
```

In "Telnet" on page 455, another example uses Telnet to do the same.

Networking Patterns

You can build networking applications from some basic patterns:

- The most common pattern is *request-reply*, also known as *request-response* or *client-server*. This pattern is synchronous: the client waits until the server responds. You've seen many examples of request-reply in this book. Your web browser is also a client, making an HTTP request to a web server, which returns a reply.
- Another common pattern is *push*, or *fanout*: you send data to any available worker in a pool of processes. An example is a web server behind a load balancer.
- The opposite of push is *pull*, or *fanin*: you accept data from one or more sources. An example is a logger that takes text messages from multiple processes and writes them to a single logfile.
- One pattern is similar to radio or television broadcasting: *publish-subscribe*, or *pub-sub*. With this pattern, a publisher sends out data. In a simple pub-sub system, all subscribers receive a copy. More often, subscribers can indicate that they're interested in only certain types of data (often called a *topic*), and the publisher will send just those. So, unlike the push pattern, more than one subscriber might receive a given piece of data. If a topic has no subscriber, the data is ignored.

Let's show some request-reply examples, and later some pub-sub ones.

The Request-Reply Pattern

Request-reply is the most familiar pattern. You request DNS, web, or email data from the appropriate servers, and they reply or tell you whether there's a problem.

I just showed you how to make basic requests with UDP or TCP, but building a networking application at the socket level is hard. Let's see if ZeroMQ can help.

Request-Reply: ZeroMQ

ZeroMQ is a library, not a server. Sometimes described as *sockets on steroids*, ZeroMQ sockets do the things that you sort of expect plain sockets to do:

- Exchange entire messages
- Retry connections
- Buffer data to preserve it when the timing between senders and receivers doesn't line up

The online guide is well written and witty, and it presents the best description of networking patterns that I've seen. The printed version (*ZeroMQ: Messaging for Many Applications*, by the late Pieter Hintjens, from that animal house, O'Reilly) has that good code smell and a big fish on the cover, rather than the other way around. All the examples in the printed guide are in the C language, but the online version lets you choose from multiple languages for each code example. The Python examples are also viewable. In this chapter, I show you some basic request-reply ZeroMQ examples.

ZeroMQ is like a LEGO set, and we all know that you can build an amazing variety of structures from a few Lego shapes. In this case, you construct networks from a few socket types and patterns. The basic "LEGO pieces" presented in the following list are the ZeroMQ socket types, which by some twist of fate look like the network patterns we've already discussed:

- `REQ` (synchronous request)
- `REP` (synchronous reply)
- `DEALER` (asynchronous request)
- `ROUTER` (asynchronous reply)
- `PUB` (publish)
- `SUB` (subscribe)
- `PUSH` (fanout)
- `PULL` (fanin)

To try these yourself, you'll need to install the Python ZeroMQ library by typing this command:

```
$ pip install pyzmq
```

The simplest pattern is a single request-reply pair. This is synchronous: one socket makes a request, and then the other replies. First, the code for the reply (server), *zmq_server.py*, is shown in Example 22-5.

Example 22-5. zmq_server.py

```
import zmq

host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
while True:
    #  Wait for next request from client
    request_bytes = server.recv()
    request_str = request_bytes.decode('utf-8')
    print("That voice in my head says: %s" % request_str)
    reply_str = "Stop saying: %s" % request_str
    reply_bytes = bytes(reply_str, 'utf-8')
    server.send(reply_bytes)
```

We create a `Context` object: this is a ZeroMQ object that maintains state. Then we make a ZeroMQ `socket` of type `REP` (for *reply*). We call `bind()` to make the socket listen on a particular IP address and port. Notice that they're specified in a string such as `tcp://localhost:6789` rather than a tuple, as in the plain-socket examples.

This example keeps receiving requests from a sender and sending a response. The messages can be very long; ZeroMQ takes care of the details.

Example 22-6 shows the code for the corresponding request (client), *zmq_client.py*. Its type is `REQ` (for *request*), and it calls `connect()` rather than `bind()`.

Example 22-6. zmq_client.py

```
import zmq

host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
for num in range(1, 6):
    request_str = "message #%s" % num
    request_bytes = request_str.encode('utf-8')
    client.send(request_bytes)
    reply_bytes = client.recv()
    reply_str = reply_bytes.decode('utf-8')
    print("Sent %s, received %s" % (request_str, reply_str))
```

Now it's time to start the client and server. One interesting difference from the plain-socket examples is that you can start them in either order. Go ahead and start the server in one window in the background:

```
$ python zmq_server.py &
```

Start the client in the same window:

```
$ python zmq_client.py
```

You'll see these alternating output lines from the client and server:

```
That voice in my head says 'message #1'
Sent 'message #1', received 'Stop saying message #1'
That voice in my head says 'message #2'
Sent 'message #2', received 'Stop saying message #2'
That voice in my head says 'message #3'
Sent 'message #3', received 'Stop saying message #3'
That voice in my head says 'message #4'
Sent 'message #4', received 'Stop saying message #4'
That voice in my head says 'message #5'
Sent 'message #5', received 'Stop saying message #5'
```

Our client ends after sending its fifth message, but we don't tell the server to quit, so it sits by the phone, waiting for another message. If you run the client again, it will print the same five lines, and the server will print its five also. If you don't kill the *zmq_server.py* process and try to run another one, Python will complain that the address is already is use:

```
$ python zmq_server.py

[2] 356
Traceback (most recent call last):
  File "zmq_server.py", line 7, in <module>
    server.bind("tcp://%s:%s" % (host, port))
  File "socket.pyx", line 444, in zmq.backend.cython.socket.Socket.bind
      (zmq/backend/cython/socket.c:4076)
  File "checkrc.pxd", line 21, in zmq.backend.cython.checkrc._check_rc
      (zmq/backend/cython/socket.c:6032)
zmq.error.ZMQError: Address already in use
```

The messages need to be sent as byte strings, so we encode our example's text strings in UTF-8 format. You can send any kind of message you like, as long as you convert it to `bytes`. We use simple text strings as the source of our messages, so `encode()` and `decode()` are enough to convert to and from byte strings. If your messages have other data types, you can use a library such as MessagePack.

Even this basic `REQ-REP` pattern allows for fancy communication patterns, because any number of `REQ` clients can `connect()` to a single `REP` server. The server handles requests one at a time, synchronously, but doesn't drop other requests that are arriving in the meantime. ZeroMQ buffers messages, up to a specified limit, until they can

get through; that's where it earns the *Q* in its name. The *Q* stands for *queue*, the *M* stands for *message*, and the *zero* means there doesn't need to be any broker.

Although ZeroMQ doesn't impose any central brokers (intermediaries), you can build them where needed. For example, use `DEALER` and `ROUTER` sockets to connect multiple sources and/or destinations asynchronously.

Multiple `REQ` sockets connect to a single `ROUTER`, which passes each request to a `DEALER`, which then contacts any `REP` sockets that have connected to it (Figure 22-1). This is similar to a bunch of browsers contacting a proxy server in front of a web server farm. This pattern lets you add multiple clients and servers as needed.

The `REQ` sockets connect only to the `ROUTER` socket; the `DEALER` connects to the multiple `REP` sockets behind it. ZeroMQ takes care of the nasty details, ensuring that the requests are load balanced and that the replies go back to the right place. Figure 22-1 shows how a broker connects clients and services.

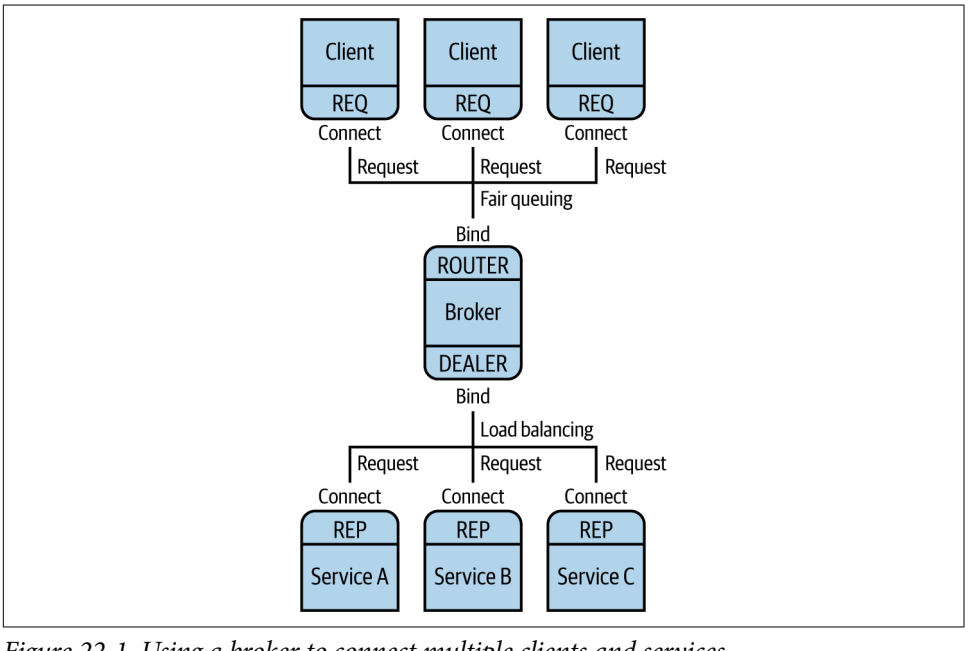

Figure 22-1. Using a broker to connect multiple clients and services

Another networking pattern called the *ventilator* uses `PUSH` sockets to farm out asynchronous tasks, and `PULL` sockets to gather the results.

The last notable feature of ZeroMQ is that it scales up *and* down, just by changing the connection type of the socket when it's created:

- `tcp` between processes, on one or more machines
- `ipc` between processes on one machine
- `inproc` between threads in a single process

That last one, `inproc`, is a way to pass data between threads without locks, and an alternative to the `threading` example in "Threads" on page 359.

After using ZeroMQ, you may not want to write raw socket code again.

Request-Reply: Other Messaging Tools

ZeroMQ is certainly not the only message-passing library that Python supports. Message passing is one of the most popular ideas in networking, and Python keeps up with other languages:

- The Apache project, whose web server we'll see in "Apache" on page 467, also maintains the ActiveMQ project, including several Python interfaces using the Simple Text Oriented Messaging Protocol (STOMP) protocol.
- RabbitMQ is also popular, and it has useful online Python tutorials.
- NATS is a fast messaging system, written in Go.

The Publish-Subscribe Pattern

Publish-subscribe (pub-sub) is not a queue but a broadcast. One or more processes publish messages. Each subscriber process indicates the type of messages it would like to receive. A copy of each message is sent to each subscriber that matches its type. Thus, a given message might be processed once, more than once, or not at all. Like a lonely ham radio operator, each publisher is just broadcasting and doesn't know who, if anyone, is listening.

Pub-Sub: Redis

In Chapter 23, Redis is mainly a data structure server, but it also contains a pub-sub system. The publisher emits messages with a topic and a value, and subscribers say which topics they want to receive.

Example 22-7 contains a publisher, *redis_pub.py*.

Example 22-7. redis_pub.py

```
import redis
import random

conn = redis.Redis()
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
for msg in range(10):
    cat = random.choice(cats)
    hat = random.choice(hats)
    print('Publish: %s wears a %s' % (cat, hat))
    conn.publish(cat, hat)
```

Each topic is a breed of cat, and the accompanying message is a type of hat.

Example 22-8 shows a single subscriber, *redis_sub.py*.

Example 22-8. redis_sub.py

```
import redis
conn = redis.Redis()

topics = ['maine coon', 'persian']
sub = conn.pubsub()
sub.subscribe(topics)
for msg in sub.listen():
    if msg['type'] == 'message':
        cat = msg['channel']
        hat = msg['data']
        print('Subscribe: %s wears a %s' % (cat, hat))
```

This subscriber wants all messages for cat types `maine coon` and `persian`, and no others. The `listen()` method returns a dictionary. If the dictionary's type is `message`, it was sent by the publisher and matches our criteria. The `channel` key is the topic (cat), and the `data` key contains the message (hat).

If you start the publisher first and no one is listening, it's like a mime falling in the forest (does he make a sound?), so start the subscriber first:

```
$ python redis_sub.py
```

Next, start the publisher. It will send 10 messages and then quit:

```
$ python redis_pub.py
Publish: maine coon wears a stovepipe
Publish: norwegian forest wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: siamese wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
```

```
Publish: maine coon wears a bowler
Publish: persian wears a bowler
Publish: norwegian forest wears a bowler
Publish: maine coon wears a stovepipe
```

The subscriber cares about only two types of cat:

```
$ python redis_sub.py
Subscribe: maine coon wears a stovepipe
Subscribe: maine coon wears a bowler
Subscribe: maine coon wears a bowler
Subscribe: persian wears a bowler
Subscribe: maine coon wears a stovepipe
```

We didn't tell the subscriber to quit, so it's still waiting for messages. If you restart the publisher, the subscriber will grab a few more messages and print them.

You can have as many subscribers (and publishers) as you want. If a message has no subscriber, the message disappears from the Redis server. However, if there are subscribers, the messages stay in the server until all subscribers have retrieved them.

Pub-Sub: ZeroMQ

ZeroMQ has no central server, so each publisher writes to all subscribers. The publisher, *zmq_pub.py*, is provided in Example 22-9.

Example 22-9. zmq_pub.py

```
import zmq
import random
import time
host = '*'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
time.sleep(1)
for msg in range(10):
    cat = random.choice(cats)
    cat_bytes = cat.encode('utf-8')
    hat = random.choice(hats)
    hat_bytes = hat.encode('utf-8')
    print('Publish: %s wears a %s' % (cat, hat))
    pub.send_multipart([cat_bytes, hat_bytes])
```

Notice that this code uses UTF-8 encoding for the topic and value strings.

The file for the subscriber is *zmq_sub.py* (Example 22-10).

Example 22-10. zmq_sub.py

```
import zmq
host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
sub = ctx.socket(zmq.SUB)
sub.connect('tcp://%s:%s' % (host, port))
topics = ['maine coon', 'persian']
for topic in topics:
    sub.setsockopt(zmq.SUBSCRIBE, topic.encode('utf-8'))
while True:
    cat_bytes, hat_bytes = sub.recv_multipart()
    cat = cat_bytes.decode('utf-8')
    hat = hat_bytes.decode('utf-8')
    print('Subscribe: %s wears a %s' % (cat, hat))
```

In this code, we subscribe to two byte values: the two strings in `topics`, encoded as UTF-8.

It seems a little backward, but if you want *all* topics, you need to subscribe to the empty byte string `b''`; if you don't, you'll get nothing.

Notice that we call `send_multipart()` in the publisher and `recv_multipart()` in the subscriber. This makes it possible for us to send multipart messages and use the first part as the topic. We could also send the topic and message as a single string or byte string, but keeping cats and hats separate seems cleaner.

Start the subscriber:

```
$ python zmq_sub.py
```

Start the publisher. It immediately sends 10 messages and then quits:

```
$ python zmq_pub.py
Publish: norwegian forest wears a stovepipe
Publish: siamese wears a bowler
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: maine coon wears a tam-o-shanter
Publish: maine coon wears a stovepipe
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: norwegian forest wears a bowler
Publish: maine coon wears a bowler
```

The subscriber prints the data it requested and received:

```
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a tam-o-shanter
Subscribe: maine coon wears a stovepipe
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a bowler
```

Pub-Sub: Other Tools

You might like to explore some of these other Python pub-sub links:

- RabbitMQ is a well-known messaging broker, and Pika is a Python API for it. See the Pika documentation and a pub-sub tutorial.
- Go to the PyPI search window and type **`pubsub`** to find Python packages like PyPubSub.
- PubSubHubbub enables subscribers to register callbacks with publishers.
- NATS is a fast, open source messaging system that supports pub-sub, request-reply, and queuing.

Internet Services

Python has an extensive networking toolset. In the following sections, we look at ways to automate some of the most popular internet services. The official, comprehensive documentation is available online.

Domain Name System

Computers have numeric IP addresses such as `85.2.101.94`, but we remember names better than numbers. The *Domain Name System* (*DNS*) is a critical internet service that converts IP addresses to and from names via a distributed database. Whenever you're using a web browser and suddenly see a message like "looking up host," you've probably lost your internet connection, and your first clue is a DNS failure.

Some DNS functions are found in the low-level `socket` module. The `gethostbyname()` function returns the IP address for a domain name, and the extended edition `gethostbyname_ex()` returns the name, a list of alternative names, and a list of addresses:

```
>>> import socket
>>> socket.gethostbyname('www.crappytaxidermy.com')
'66.6.44.4'
>>> socket.gethostbyname_ex('www.crappytaxidermy.com')
('crappytaxidermy.com', ['www.crappytaxidermy.com'], ['66.6.44.4'])
```

The `getaddrinfo()` method looks up the IP address, but it also returns enough information to create a socket to connect to it:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80)
[(2, 2, 17, '', ('66.6.44.4', 80)),
(2, 1, 6, '', ('66.6.44.4', 80))]
```

The preceding call returns two tuples: the first for UDP, and the second for TCP (the `6` in the `2, 1, 6` is the value for TCP).

You can ask for TCP or UDP information only:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80, socket.AF_INET,
socket.SOCK_STREAM)
[(2, 1, 6, '', ('66.6.44.4', 80))]
```

Some TCP and UDP port numbers are reserved for certain services by IANA, and are associated with service names. For example, HTTP is named `http` and is assigned TCP port 80.

These functions convert between service names and port numbers:

```
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyport(80)
'http'
```

Python Email Modules

The standard library contains these email modules:

- smtplib for sending email messages via Simple Mail Transfer Protocol (SMTP)
- email for creating and parsing email messages
- poplib for reading email via Post Office Protocol 3 (POP3)
- imaplib for reading email via Internet Message Access Protocol (IMAP)

If you want to write your own Python SMTP server, try the new asynchronous version aiosmtpd.

Other Protocols

Using the standard `ftplib` module, you can push bytes around by using the File Transfer Protocol (FTP). Although it's an old protocol, FTP still performs well.

You've seen many of these modules in various places in this book, but also try the documentation for standard library support of internet protocols.

Web Services and APIs

Information providers always have a website, but those are targeted for human eyes, not automation. If data is published only on a website, anyone who wants to access and structure the data needs to write scrapers (as shown in "Crawl and Scrape" on page 484), and rewrite them each time a page format changes. This is usually tedious. In contrast, if a website offers an API to its data, the data becomes directly available to client programs. APIs change less often than web page layouts, so client rewrites are less common. A fast, clean data pipeline also makes it easier to build unforeseen but useful combinations.

In many ways, the easiest API is a web interface, but one that provides data in a structured format such as JSON or XML rather than plain text or HTML. The API might be minimal or a full-fledged RESTful API (defined in "Web APIs and REST" on page 480), but it provides another outlet for those restless bytes.

In Chapter 1, you saw a web API query the Internet Archive for an old copy of a website.

APIs are especially useful for mining well-known social media sites such as Twitter/X, Facebook/Meta, and LinkedIn. All these sites provide APIs that are free to use, but they require you to register and get a *key* (a long, generated text string, also known as a *token*) to use when connecting. The key lets a site determine who's accessing its data. It can also serve as a way to limit request traffic to servers.

Here are some interesting service APIs:

- New York Times
- Twitter
- Facebook
- Weather Underground
- Marvel Comics

Data Serialization

As you'll see in Chapter 23, formats like XML, JSON, and YAML are ways to store structured text data. Networked applications need to exchange data with other programs. The conversion between data in memory and byte sequences "on the wire" is called *serialization*, or *marshaling*. JSON is a popular serialization format, especially with web RESTful systems, but it can't express all Python data types directly. Also, as a text format it tends to be more verbose than some binary serialization methods. Let's look at some approaches that you're likely to run into.

Serialize with pickle

Python provides the `pickle` module to save and restore any object in a special binary format.

Remember how JSON lost its mind when encountering a `datetime` object? Not a problem for `pickle`:

```
>>> import pickle
>>> import datetime
>>> now1 = datetime.datetime.now(datetime.UTC)
>>> pickled = pickle.dumps(now1)
>>> now2 = pickle.loads(pickled)
>>> now1
datetime.datetime(2025, 2, 17, 4, 3, 45, 291365)
>>> now2
datetime.datetime(2025, 2, 17, 4, 3, 45, 291365))
```

The `pickle` module works with your own classes and objects too. Let's define a little class called `Tiny` that returns the string `tiny` when treated as a string:

```
>>> import pickle
>>> class Tiny():
...     def __str__(self):
...         return 'tiny'
...
>>> obj1 = Tiny()
>>> obj1
<__main__.Tiny object at 0x10076ed10>
>>> str(obj1)
'tiny'
>>> pickled = pickle.dumps(obj1)
>>> pickled
b'\x80\x03c__main__\nTiny\nq\x00)\x81q\x01.'
>>> obj2 = pickle.loads(pickled)
>>> obj2
<__main__.Tiny object at 0x10076e550>
>>> str(obj2)
'tiny'
```

`pickled` is the pickled binary string made from the object `obj1`. We converted that back to the object `obj2` to make a copy of `obj1`. Use `dump()` to pickle to a file, and `load()` to unpickle from one.

The `multiprocessing` module uses `pickle` to interchange data among processes.

If `pickle` can't serialize your data format, a newer third-party package called dill might.

Because `pickle` can create Python objects, the same security warnings that were discussed in earlier sections apply. A public service announcement: don't unpickle something that you don't trust.

Use Other Serialization Formats

These binary data interchange formats are usually more compact and faster than XML or JSON:

- MessagePack—aka MsgPack
- msgspec
- Protocol Buffers
- Apache Avro
- Apache Thrift
- lima
- Serialize
- A benchmark of various Python serialization packages

Because they are binary, none can be easily edited by a human with a text editor.

Some third-party packages interconvert objects and basic Python data types (allowing further conversion to/from formats like JSON), and provide *validation* of the following:

- Data types
- Value ranges
- Required versus optional data

These include the following:

- marshmallow
- Pydantic—uses type hints, so requires at least Python 3.6
- TypeSystem

These are often used with web servers to ensure that the bytes that came over the wire via HTTP end up in the right data structures for further processing.

Remote Procedure Calls

Remote procedure calls (*RPCs*) look like normal functions but execute on remote machines across a network. Instead of calling a RESTful API with arguments encoded in the URL or request body, you call an RPC function on your own machine. Your local machine does the following:

- Serializes your function arguments into bytes
- Sends the encoded bytes to the remote machine

The remote machine takes these steps:

- Receives the encoded request bytes
- Deserializes the bytes back to data structures
- Finds and calls the service function with the decoded data
- Encodes the function results
- Sends the encoded bytes back to the caller

And finally, the local machine that started it all does this:

- Decodes the bytes to return values

RPC is a popular technique, and people have implemented it in many ways. On the server side, you start a server program, connect it with a byte transport and encoding/decoding method, define service functions, and light up your *RPC is open for business* sign. The client connects to the server and calls one of its functions via RPC.

XML-RPC

The standard library includes one RPC implementation that uses XML as the exchange format: XML-RPC. You define and register functions on the server, and the client calls them as though they were imported. First, let's explore the file *xmlrpc_server.py* as shown in Example 22-11.

Example 22-11. xmlrpc_server.py

```
from xmlrpc.server import SimpleXMLRPCServer

def double(num):
    return num * 2

server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(double, "double")
server.serve_forever()
```

The function we're providing on the server is called `double()`. It expects a number as an argument and returns the value of that number times two. The server starts up on an address and port. We need to *register* the function to make it available to clients via RPC. Finally, we start serving and carry on.

Now—you guessed it—*xmlrpc_client.py*, proudly presented in Example 22-12.

Example 22-12. xmlrpc_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
num = 7
result = proxy.double(num)
print("Double %s is %s" % (num, result))
```

The client connects to the server by using `ServerProxy()`. Then the client calls the function `proxy.double()`. Where did that come from? It was created dynamically by the server. The RPC machinery magically hooks this function name into a call to the remote server.

Give it a try—start the server and then run the client:

```
$ python xmlrpc_server.py
```

Run the client again:

```
$ python xmlrpc_client.py
Double 7 is 14
```

The server then prints the following:

```
127.0.0.1 - - [31/Jul/2025 16:45:46] "POST / HTTP/1.1" 200 -
```

Popular transport methods are HTTP and ZeroMQ.

JSON RPC

JSON-RPC (versions 1.0 and 2.0) is similar to XML-RPC, but with JSON. There are many Python JSON-RPC libraries, but the simplest one I've found comes in two parts: client and server.

Installation of both is familiar: `pip install jsonrpcserver` and `pip install jsonrpclient`.

These libraries provide many alternative ways to write a client and server. In Examples 22-13 and 22-14, I use this library's built-in server, which uses port 5000 and is the simplest.

First, here's the server.

Example 22-13. jsonrpc_server.py

```
from jsonrpcserver import method, serve

@method
def double(num):
    return num * 2

if __name__ == "__main__":
    serve()
```

Second, here's the client.

Example 22-14. jsonrpc_client.py

```
from jsonrpcclient import request

num = 7
response = request("http://localhost:5000", "double", num=num)
print("Double", num, "is", response.data.result)
```

As with most of the client-server examples in this chapter, start the server first (in its own terminal window, or with a following `&` to put it in the background) and then run the client:

```
$ python jsonrpc_server.py &
[1] 10621
$ python jsonrpc_client.py
127.0.0.1 - - [23/Jun/2019 15:39:24] "POST / HTTP/1.1" 200 -
Double 7 is 14
```

MessagePack RPC

The encoding library MessagePack has its own Python RPC implementation. Install it with `pip install msgpack-rpc-python`.

This will also install Tornado, a Python event-based web server that this library uses as a transport. As usual, the server (*msgpack_server.py*) comes first (Example 22-15).

Example 22-15. msgpack_server.py

```
from msgpackrpc import Server, Address

class Services():
    def double(self, num):
        return num * 2
```

```
server = Server(Services())
server.listen(Address("localhost", 6789))
server.start()
```

The `Services` class exposes its methods as RPC services. Go ahead and start the client, *msgpack_client.py* (Example 22-16).

Example 22-16. msgpack_client.py

```
from msgpackrpc import Client, Address

client = Client(Address("localhost", 6789))
num = 8
result =  client.call('double', num)
print("Double %s is %s" % (num, result))
```

To run the server and client, follow the usual drill: start the server and client in separate terminal windows and observe the results:[1]

```
$ python msgpack_server.py

$ python msgpack_client.py
Double 8 is 16
```

zerorpc

Written by the developers of Docker (when it was called dotCloud), zerorpc uses ZeroMQ and MessagePack to connect clients and servers. It magically exposes functions as RPC endpoints.

Type **`pip install zerorpc`** to install it. The sample code in Examples 22-17 and 22-18 shows a request-reply client and server.

Example 22-17. zerorpc_server.py

```
import zerorpc

class RPC():
    def double(self, num):
        return 2 * num

server = zerorpc.Server(RPC())
server.bind("tcp://0.0.0.0:4242")
server.run()
```

1 Or put the server in the background with a final `&`.

Example 22-18. zerorpc_client.py

```
import zerorpc

client = zerorpc.Client()
client.connect("tcp://127.0.0.1:4242")
num = 7
result = client.double(num)
print("Double", num, "is", result)
```

Notice that the client calls `client.double()`, even though there's no definition of it in there:

```
$ python zerorpc_server &
[1] 55172
$ python zerorpc_client.py
Double 7 is 14
```

zerorpc.io has many more examples.

gRPC

Google created gRPC as a portable and fast way to define and connect services. It encodes data as Protocol Buffers.

Install the Python parts with `pip install grpcio` and `pip install grpcio-tools`.

The Python client docs are detailed, so I'm giving only a brief overview here.

To use gRPC, you write a *.proto* file to define a `service` and its `rpc` methods.

An `rpc` method is like a function definition (describing its arguments and return types) and may specify one of these networking patterns:

- Request-response (sync or async)
- Request-streaming response
- Streaming request-response (sync or async)
- Streaming request-streaming response

Single responses can be blocking or asynchronous. Streaming responses are iterated.

Next, you would run the `grpc_tools.protoc` program to create Python code for the client and server. gRPC handles the serialization and network communication; you add your application-specific code to the client and server stubs.

gRPC is a top-level alternative to web REST APIs. It seems to be a better fit than REST for interservice communication, and REST may be preferred for public APIs.

Remote Management

A category of tools can manage multiple remote processes:

- Salt is written in Python. It started as a way to implement remote execution but grew to a full-fledged systems management platform. Based on ZeroMQ rather than SSH, it can scale to thousands of servers.
- Puppet and Chef are popular and closely tied to Ruby.
- The Ansible package, which like Salt is written in Python, is also comparable. It's free to download and use, but support and some add-on packages require a commercial license. Ansible uses SSH by default and does not require any special software to be installed on the machines that it will manage.

Big Fat Data

As Google and other internet companies grew, they found that traditional computing solutions didn't scale. Software that worked for single machines, or even a few dozen, could not keep up with thousands.

Disk storage for databases and files involved too much *seeking*, which requires mechanical movement of disk heads. (Think of a vinyl record, and the time it takes to manually move the needle from one track to another. And think of the screeching sound the needle makes when you drop it too hard, not to mention the sounds made by the record's owner.) But you could *stream* consecutive segments of the disk more quickly.

Developers found that it was faster to distribute and analyze data on many networked machines than on individual ones. They could use algorithms that sounded simplistic but actually worked better overall with massively distributed data. One of these is MapReduce, which spreads a calculation across many machines and then gathers the results. This process is similar to working with queues.

Hadoop

After Google published its MapReduce results in a paper, Yahoo followed with an open source Java-based package called Apache Hadoop (named after the toy stuffed elephant of the lead programmer's son). For several years, if *big data* was mentioned, the response for many organizations was Hadoop.

"Big data" often just means "data too big to fit on my machine": disk, memory, CPU time, or all of the above.

Hadoop copies data among machines, running them through *map* (scatter) and *reduce* (gather) programs, and saving the results on disk at each step.

This batch process can be slow. A quicker method called *Hadoop streaming* works like Unix pipes, streaming the data through programs without requiring disk writes at each step. You can write Hadoop streaming programs in any language, including Python.

Many Python modules have been written for Hadoop, and some are discussed in the article "Hadoop with Python" by David Adrián Cañones. Spotify, known for streaming music, released its Python component for Hadoop streaming, Luigi, as open source.

Spark

A rival named Apache Spark was designed to run 10 to 100 times faster than Hadoop. It can read and process any Hadoop data source and format. Spark includes APIs for Python and other languages, and uses AI and machine learning. You can find the installation documents online.

Disco

Another alternative to Hadoop is Disco, which uses Python for MapReduce processing and Erlang for communication. It looks like the code hasn't been to touched for ten years, but if you're curious, see the documentation.

Dask

Dask is similar to Spark, although it's written in Python and is largely used with scientific Python packages like NumPy, pandas, and scikit-learn. Dask can spread tasks across thousand-machine clusters.

To get Dask and all its extra helpers, use `pip install dask[complete]`.

See Chapter 23 for related examples of *parallel programming*, in which a large structured calculation is distributed among many machines.

Clouds

> *I really don't know clouds at all.*
>
> —Joni Mitchell

Not so long ago, you would buy your own servers, bolt them into racks in data centers, and install layers of software on them: operating systems, device drivers, filesystems, databases, web servers, email servers, name servers, load balancers, monitors, and more. Any initial novelty wore off as you tried to keep multiple systems alive and responsive. And you worried constantly about security.

Many hosting services offered to take care of your servers for a fee, but you still leased the physical devices and had to pay for your peak load configuration at all times.

With more individual machines, failures are no longer infrequent: they're very common. You need to scale services horizontally and store data redundantly. You can't assume that the network operates like a single machine. The eight fallacies of distributed computing, according to Peter Deutsch, are as follows:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

You can try to build these complex distributed systems, but it's a lot of work, and a different toolset is needed. To borrow an analogy, when you have a handful of servers, you treat them like pets: you give them names, know their personalities, and nurse them back to health when needed. But at scale, you treat servers more like livestock: they look alike, have numbers, and are replaced if they have any problems.

Instead of building, you can rent servers in the *cloud*. By adopting this model, maintenance is someone else's problem, and you can concentrate on your service, or blog, or whatever you want to show the world. Using web dashboards and APIs, you can spin up servers with whatever configuration you need, quickly and easily—they're *elastic*. You can monitor their status, and be alerted if a metric exceeds a given threshold. Clouds are currently a pretty hot topic, and corporate spending on cloud components has spiked.

The big cloud vendors are listed here:

- Amazon
- Google
- Microsoft Azure

Amazon Web Services

As Amazon was growing from hundreds to thousands to millions of servers, developers ran into all the nasty problems of distributed systems. One day in 2002 or thereabouts, CEO Jeff Bezos declared to Amazon employees that, henceforth, all data and functionality needed to be exposed only via network service interfaces—not files, or databases, or local function calls. They had to design these interfaces as though they were being offered to the public. The memo ended with a motivational nugget: "Anyone who doesn't do this will be fired."

Not surprisingly, developers got to work, and over time built a huge service-oriented architecture. They borrowed or innovated many solutions, evolving into Amazon Web Services (AWS), which now dominates the market. The official Python AWS library is Boto3, which you can read about in the following resources:

- Documentation
- SDK pages

Install Boto3 with `pip install boto3`.

You can use Boto3 as an alternative to the AWS web-based management pages.

Google Cloud

Google uses Python a lot internally, and it employs some prominent Python developers (even Guido van Rossum himself, for some time). From the Google Cloud main and Python pages, you can find details on its many services.

Microsoft Azure

Microsoft caught up with Amazon and Google with its cloud offering, Azure. See "Python on Azure" at the Azure website to learn how to develop and deploy Python applications.

OpenStack

OpenStack is an open source framework of Python services and REST APIs. Many of its services are similar to those in the commercial clouds.

Docker

The humble standardized shipping container revolutionized international trade. Only a few years ago, Docker applied the *container* name and analogy to a *virtualization* method using little-known Linux features. Containers are much lighter than virtual machines, and a bit heavier than Python virtualenvs. They allow you to package an

application separately from other applications on the same machine, sharing only the operating system kernel.

To install Docker's Python client library, use `pip install docker`.

Kubernetes

Containers caught on and spread through the computing world. Eventually, people needed ways to manage multiple containers and wanted to automate some of the manual steps that have been usually required in large distributed systems:

- Failover
- Load balancing
- Scaling up and down

It looks like Kubernetes is leading the pack in this new area of *container orchestration*.

To install the Python client library, use `pip install kubernetes`.

Review/Preview

This was another heavy chapter. It discussed many ways to ship data around, from basic byte-pushing to service management.

From data galloping freely over networks, we go to data stuck in a box: databases and other persistent storage methods.

Practice

22.1 Use a plain `socket` to implement a current-time-service. When a client sends the string `time` to the server, return the current date and time as an ISO string.

22.2 Use ZeroMQ `REQ` and `REP` sockets to do the same thing.

22.3 Try the same with XML-RPC.

22.4 You may have seen the classic *I Love Lucy* television episode in which Lucy and Ethel work in a chocolate factory. The duo falls behind as the conveyor belt that supplies the confections for them to process begins operating at an ever-faster rate. Write a simulation that pushes different types of chocolates to a Redis list, and Lucy is a client doing Redis "blocking pops" of this list. She needs 0.5 seconds to handle a piece of chocolate. Print the time and type of each chocolate as Lucy gets it, as well as the number that remain to be handled.

CHAPTER 23

Data in a Box: Persistent Storage

> *One thing a computer can do that most humans can't is be sealed up in a cardboard box and sit in a warehouse.*
>
> —Jack Handey

An active program accesses data stored in random access memory, or RAM. RAM is very fast, but it is expensive and requires a constant supply of power; if the power goes out, all the data in memory is lost. Disk drives are slower than RAM but have more capacity, cost less, and retain data even after someone trips over the power cord. Thus, a huge amount of effort in computer systems has been devoted to making the best trade-offs between storing data on disk and RAM. As programmers, we need *persistence*: storing and retrieving data using nonvolatile media such as disks.

This chapter is all about the different flavors of data storage, each optimized for different purposes: flat files, structured files, and databases. File operations like input and output are covered in Chapter 20.

Text Files

The simplest persistence is a plain old *flat text file*. This works well if your data has a simple structure and you exchange all of it between disk and memory. Plain text data might be suitable for this treatment.

A *record* is a term for one chunk of related data, consisting of individual *fields*.

In a *padded text file*, each field in a record has a fixed width, and the data is padded (usually with space characters) to that width in the file, giving each line (record) the same width. A programmer can use `seek()` to jump around the file and read and write only the records and fields that are needed.

Tabular and Delimited Text Files

With simple text files, the only level of organization is the line. Sometimes you want more structure than that. You might want to save data for your program to use later, or send data to another program.

Text files can have many formats, and here's how you can distinguish them:

- A *separator*, or *delimiter*, character like a tab (`\t`), comma (`,`), or vertical bar (`|`). This is an example of the comma-separated values (CSV) format.
- < and > around *tags*. Examples include XML and HTML.
- Punctuation. An example is JavaScript Object Notation (JSON).
- Indentation. An example is YAML (which is recursively defined as *YAML Ain't Markup Language*).
- Miscellaneous, such as configuration files for programs.

Each of these structured file formats can be read and written by at least one Python module.

CSV

Delimited files are often used as an exchange format for spreadsheets and databases. You could read CSV files manually, a line at a time, splitting each line into fields at comma separators, and adding the results to data structures such as lists and dictionaries. But it's better to use the standard `csv` module, because parsing these files can get more complicated than you think. Following are a few important characteristics to keep in mind when working with CSV:

- Some have alternate delimiters besides a comma; `|` and `\t` (tab) are common.
- Some have *escape sequences*. If the delimiter character can occur within a field, the entire field might be surrounded by quote characters or preceded by an escape character.
- Files have different line-ending characters. Unix uses `\n`, Microsoft uses `\r\n`, and Apple used to use `\r` but now uses `\n`.
- The first line may contain column names.

First, we see how to read and write a list of rows, each containing a list of columns:

```
>>> import csv
>>> villains = [
...     ['Doctor', 'No'],
...     ['Rosa', 'Klebb'],
...     ['Mister', 'Big'],
```

```
...     ['Auric', 'Goldfinger'],
...     ['Ernst', 'Blofeld'],
...     ]
>>> with open('villains', 'wt') as fout:
...     csvout = csv.writer(fout)
...     csvout.writerows(villains)
```

That `with open` line is a *context manager* that automatically closes the file when its code block ends; see "Close Files Automatically by Using with" on page 335.

This creates the file *villains* with these lines:

```
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Now, we try to read the file back in:

```
>>> import csv
>>> with open('villains', 'rt') as fin:  # context manager
...     cin = csv.reader(fin)
...     villains = [row for row in cin]  # a list comprehension
...
>>> print(villains)
[['Doctor', 'No'], ['Rosa', 'Klebb'], ['Mister', 'Big'],
['Auric', 'Goldfinger'], ['Ernst', 'Blofeld']]
```

We're taking advantage of the structure created by the `reader()` function. It created rows in the `cin` object that we can extract in a `for` loop.

Using `reader()` and `writer()` with their default options, the columns are separated by commas and the rows by line feeds.

The data can be a list of dictionaries rather than a list of lists. Let's read the *villains* file again, this time using the new `DictReader()` function and specifying the column names:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin, fieldnames=['first', 'last'])
...     villains = [row for row in cin]
...
>>> print(villains)
[OrderedDict([('first', 'Doctor'), ('last', 'No')]),
OrderedDict([('first', 'Rosa'), ('last', 'Klebb')]),
OrderedDict([('first', 'Mister'), ('last', 'Big')]),
OrderedDict([('first', 'Auric'), ('last', 'Goldfinger')]),
OrderedDict([('first', 'Ernst'), ('last', 'Blofeld')])]
```

That `OrderedDict` is there for compatibility with versions of Python before 3.6, when dictionaries kept their order by default.

Let's rewrite the CSV file by using the new `DictWriter()` function. We also call `writeheader()` to write an initial line of column names to the CSV file:

```
import csv
villains = [
    {'first': 'Doctor', 'last': 'No'},
    {'first': 'Rosa', 'last': 'Klebb'},
    {'first': 'Mister', 'last': 'Big'},
    {'first': 'Auric', 'last': 'Goldfinger'},
    {'first': 'Ernst', 'last': 'Blofeld'},
    ]
with open('villains.txt', 'wt') as fout:
    cout = csv.DictWriter(fout, ['first', 'last'])
    cout.writeheader()
    cout.writerows(villains)
```

That creates a *villains.csv* file with a header line (Example 23-1).

Example 23-1. villains.csv

```
first,last
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Now, let's read back the file. By omitting the `fieldnames` argument in the `DictReader()` call, we tell it to use the values in the first line of the file (`first,last`) as column labels and matching dictionary keys:

```
>>> import csv
>>> with open('villains.csv', 'rt') as fin:
...     cin = csv.DictReader(fin)
...     villains = [row for row in cin]
...
>>> print(villains)
[OrderedDict([('first', 'Doctor'), ('last', 'No')]),
OrderedDict([('first', 'Rosa'), ('last', 'Klebb')]),
OrderedDict([('first', 'Mister'), ('last', 'Big')]),
OrderedDict([('first', 'Auric'), ('last', 'Goldfinger')]),
OrderedDict([('first', 'Ernst'), ('last', 'Blofeld')])]
```

XML

Delimited files convey only two dimensions: rows (lines) and columns (fields within a line). If you want to exchange data structures among programs, you need a way to encode hierarchies, sequences, sets, and other structures as text.

Extensible Markup Language (XML) is a prominent *markup* format that does this. It uses *tags* to delimit data, as in this sample *menu.xml* file:

```
<?xml version="1.0"?>
<menu>
  <breakfast hours="7-11">
    <item price="$6.00">breakfast burritos</item>
    <item price="$4.00">pancakes</item>
  </breakfast>
  <lunch hours="11-3">
    <item price="$5.00">hamburger</item>
  </lunch>
  <dinner hours="3-10">
    <item price="8.00">spaghetti</item>
  </dinner>
</menu>
```

Following are a few important characteristics of XML:

- Tags begin with a `<` character. The tags in this sample are `menu`, `breakfast`, `lunch`, `dinner`, and `item`.
- Whitespace is ignored.
- Usually a *start tag* such as `<menu>` is followed by other content and then a final matching *end tag* such as `</menu>`.
- Tags can *nest* within other tags to any level. In this example, `item` tags are children of the `breakfast`, `lunch`, and `dinner` tags; they, in turn, are children of `menu`.
- Optional *attributes* can occur within the start tag. In this example, `price` is an attribute of `item`.
- Tags can contain *values*. In this example, each `item` has a value, such as `pancakes` for the second breakfast item.
- If a tag named `thing` has no values or children, it can be expressed as the single tag by including a forward slash just before the closing angle bracket, such as `<thing/>`, rather than a start and end tag, like `<thing></thing>`.
- The choice of where to put data—attributes, values, child tags—is somewhat arbitrary. For instance, we could have written the last `item` tag as `<item price="$8.00" food="spaghetti"/>`.

XML is often used for data *feeds* and *messages* and has subformats like RSS and Atom. Some industries have many specialized XML formats, such as the finance field.

XML's über-flexibility has inspired multiple Python libraries that differ in approach and capabilities.

The simplest way to parse XML in Python is by using the standard `ElementTree` module. Here's a little program to parse the *menu.xml* file and print some tags and attributes:

```
>>> import xml.etree.ElementTree as et
>>> tree = et.ElementTree(file='menu.xml')
>>> root = tree.getroot()
>>> root.tag
'menu'
>>> for child in root:
...     print('tag:', child.tag, 'attributes:', child.attrib)
...     for grandchild in child:
...         print('\ttag:', grandchild.tag, 'attributes:', grandchild.attrib)
...
tag: breakfast attributes: {'hours': '7-11'}
    tag: item attributes: {'price': '$6.00'}
    tag: item attributes: {'price': '$4.00'}
tag: lunch attributes: {'hours': '11-3'}
    tag: item attributes: {'price': '$5.00'}
tag: dinner attributes: {'hours': '3-10'}
    tag: item attributes: {'price': '8.00'}
>>> len(root)     # number of menu sections
3
>>> len(root[0])  # number of breakfast items
2
```

For each element in the nested lists, `tag` is the tag string, and `attrib` is a dictionary of its attributes. `ElementTree` has many other ways of searching XML-derived data, modifying it, and even writing XML files. The `ElementTree` documentation has the details.

Other standard Python XML libraries include the following:

xml.dom
: The Document Object Model (DOM), familiar to JavaScript developers, represents web documents as hierarchical structures. This module loads the entire XML file into memory and lets you access all the pieces equally.

xml.sax
: Simple API for XML, or SAX, parses XML on the fly, so it does not have to load everything into memory at once. Therefore, it can be a good choice if you need to process very large streams of XML.

An XML Security Note

You can use all the formats described in this chapter to save objects to files and read them back again. It's possible to exploit this process and cause security problems.

For example, the following XML snippet from the "Billion Laughs Attack" Wikipedia page defines 10 nested entities, each expanding the lower level 10 times for a total expansion of one billion:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

The bad news: the billion laughs would blow up all the XML libraries mentioned in the previous sections. Defused XML lists this attack and others, along with the vulnerability of Python libraries.

The link shows how to change the settings for many of the libraries to avoid these problems. Also, you can use the defusedxml library as a security frontend for the other libraries:

```
>>> # insecure:
>>> from xml.etree.ElementTree import parse
>>> et = parse(xmlfile)
>>> # protected:
>>> from defusedxml.ElementTree import parse
>>> et = parse(xmlfile)
```

The standard Python site also has its own page on XML vulnerabilities.

HTML

Gigagobs of data are saved as *Hypertext Markup Language* (*HTML*), the basic document format of the web. The problem is that much of it doesn't follow the HTML rules, which can make it difficult to parse. HTML is a better display format than a data interchange format. Because this chapter is intended to describe fairly well-defined data formats, I've separated out the discussion about HTML to Chapter 24.

JSON

JavaScript Object Notation (JSON) has become a popular data interchange format, beyond its JavaScript origins. The JSON format is a subset of JavaScript, and often legal Python syntax as well. Its close fit to Python makes it a good choice for data

interchange among programs. You'll see many examples of JSON for web development in Chapter 24.

Unlike the variety of XML modules, there's one main JSON module, with the unforgettable name `json`. This program encodes (dumps) data to a JSON string and decodes (loads) a JSON string back to data. In this next example, let's build a Python data structure containing the data from the earlier XML example:

```
>>> menu = {
... "breakfast": {
...         "hours": "7-11",
...         "items": {
...                 "breakfast burritos": "$6.00",
...                 "pancakes": "$4.00"
...                 }
...         },
... "lunch" : {
...         "hours": "11-3",
...         "items": {
...                 "hamburger": "$5.00"
...                 }
...         },
... "dinner": {
...         "hours": "3-10",
...         "items": {
...                 "spaghetti": "$8.00"
...                 }
...         }
... }
```

Next, encode the data structure (`menu`) to a JSON string (`menu_json`) by using `dumps()`:

```
>>> import json
>>> menu_json = json.dumps(menu)
>>> menu_json
'{"dinner": {"items": {"spaghetti": "$8.00"}, "hours": "3-10"},
"lunch": {"items": {"hamburger": "$5.00"}, "hours": "11-3"},
"breakfast": {"items": {"breakfast burritos": "$6.00", "pancakes":
"$4.00"}, "hours": "7-11"}}'
```

And now, let's turn the JSON string `menu_json` back into a Python data structure (`menu2`) by using `loads()`:

```
>>> menu2 = json.loads(menu_json)
>>> menu2
{'breakfast': {'items': {'breakfast burritos': '$6.00', 'pancakes':
'$4.00'}, 'hours': '7-11'}, 'lunch': {'items': {'hamburger': '$5.00'},
'hours': '11-3'}, 'dinner': {'items': {'spaghetti': '$8.00'}, 'hours': '3-10'}}
```

The `menu` and `menu2` objects are both dictionaries with the same keys and values.

You might get an exception while trying to encode or decode some objects, including objects such as `datetime` (covered in detail in Chapter 19), as demonstrated here:

```
>>> import datetime
>>> import json
>>> now = datetime.datetime.utcnow()
>>> now
datetime.datetime(2025, 8, 1, 17, 5, 39, 43842)
>>> json.dumps(now)
Traceback (most recent call last):
# ... (deleted stack trace to save trees)
TypeError: Object of type datetime is not JSON serializable
>>>
```

This can happen because the JSON standard does not define date or time types; it expects you to define how to handle them. You could convert the `datetime` to something JSON understands, such as a string or an *epoch* value (see Chapter 19):

```
>>> now_str = str(now)
>>> json.dumps(now_str)
'"2025-08-01 17:05:39.043842"'
>>> from time import mktime
>>> now_epoch = int(mktime(now.timetuple()))
>>> json.dumps(now_epoch)
1754085939'
```

If the `datetime` value could occur in the middle of normally converted data types, making these special conversions might be annoying. You can modify the way JSON is encoded by using inheritance, which is described in Chapter 11. Python's JSON documentation gives an example of this for complex numbers, which also makes JSON play dead. Let's modify the code for `datetime`:

```
>>> import time
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> class DTEncoder(json.JSONEncoder):
...     def default(self, obj):
...         # isinstance() checks the type of obj
...         if isinstance(obj, datetime.datetime):
...             return int(time.mktime(obj.timetuple()))
...         # else it's something the normal decoder knows:
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(now, cls=DTEncoder)
'1754086614'
```

The new class `DTEncoder` is a subclass, or child class, of `JSONEncoder`. We need to override its only `default()` method to add `datetime` handling. Inheritance ensures that everything else will be handled by the parent class.

The `isinstance()` function checks whether the object `obj` is of the class `datetime.datetime`. Because everything in Python is an object, `isinstance()` works everywhere:

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> type(now)
<class 'datetime.datetime'>
>>> isinstance(now, datetime.datetime)
True
>>> type(234)
<class 'int'>
>>> isinstance(234, int)
True
>>> type('hey')
<class 'str'>
>>> isinstance('hey', str)
True
```

For JSON and other structured text formats, you can load from a file into data structures without knowing anything about the structures ahead of time. Then you can walk through the structures by using `isinstance()` and type-appropriate methods to examine their values. For example, if one of the items is a dictionary, you can extract contents through `keys()`, `values()`, and `items()`.

After making you do it the hard way, it turns out that there's an even easier way to convert `datetime` objects to JSON:

```
>>> import datetime
>>> import json
>>> now = datetime.datetime.utcnow()
>>> json.dumps(now, default=str)
'"2025-08-01 17:18:36.285613"'
```

That `default=str` tells `json.dumps()` to apply the `str()` conversion function for data types that it doesn't understand. This works because the definition of the `datetime.datetime` class includes a `__str__()` method.

YAML

Similar to JSON, YAML has keys and values, but handles more data types such as dates and times. The standard Python library does not yet include YAML handling, so you need to install a third-party library named PyYAML to manipulate it. The `load()` function converts a YAML string to Python data, whereas `dump()` does the opposite.

The following YAML file, *mcintyre.yaml*, contains information on the Canadian poet James McIntyre, including two of his poems:

```
name:
  first: James
  last: McIntyre
dates:
  birth: 1828-05-25
  death: 1906-03-31
details:
  bearded: true
  themes: [cheese, Canada]
books:
  url: http://www.gutenberg.org/files/36068/36068-h/36068-h.htm
poems:
  - title: 'Motto'
    text: |
      Politeness, perseverance and pluck,
      To their possessor will bring good luck.
  - title: 'Canadian Charms'
    text: |
      Here industry is not in vain,
      For we have bounteous crops of grain,
      And you behold on every field
      Of grass and roots abundant yield,
      But after all the greatest charm
      Is the snug home upon the farm,
      And stone walls now keep cattle warm.
```

Values such as `true`, `false`, `on`, and `off` are converted to Python Booleans. Integers and strings are converted to their Python equivalents. Other syntax creates lists and dictionaries:

```
>>> import yaml
>>> with open('mcintyre.yaml', 'rt') as fin:
>>>     text = fin.read()
>>> data = yaml.load(text)
>>> data['details']
{'themes': ['cheese', 'Canada'], 'bearded': True}
>>> len(data['poems'])
2
```

The data structures that are created match those in the YAML file, which in this case are more than one level deep in places. You can get the title of the second poem with this dict/list/dict reference:

```
>>> data['poems'][1]['title']
'Canadian Charms'
```

PyYAML can load Python objects from strings, and this is dangerous. Use `safe_load()` instead of `load()` if you're importing YAML that you don't trust. Better yet, *always* use `safe_load()`. Read Ned Batchelder's blog post "War Is Peace" for a description of how unprotected YAML loading compromised the Ruby on Rails platform.

TOML

Tom's Obvious Minimal Language (TOML) is like YAML, but easier to understand and parse. TOML supports comments and is handy for config files, and unlike YAML, has a module in the Python standard library: tomllib.

Tablib

One third-party package lets you import, export, and edit tabular data in CSV, JSON, *or* YAML format, as well as Microsoft Excel, pandas DataFrame, and a few others.[1] Install Tablib with `pip install tablib`, and look at the docs.

Configuration Files

Most programs offer various *options*, or *settings*. Dynamic ones can be provided as program arguments, but long-lasting ones need to be kept somewhere. The temptation to define your own quick-and-dirty *config file* format is strong—but resist it. The format often turns out to be dirty but not so quick. You need to maintain both the writer program and the reader program (sometimes called a *parser*). Good alternatives are available that you can just drop into your program, including those in the previous sections.

Here, we'll use the standard `configparser` module, which handles Windows-style *.ini* files. Such files have sections of *`key`* = *`value`* definitions. Here's a minimal *settings.cfg* file:

```
[english]
greeting = Hello

[french]
greeting = Bonjour

[files]
home = /usr/local
# simple interpolation:
bin = %(home)s/bin
```

1 Alas, not XML yet.

Here's the code to read the file into Python data structures:

```
>>> import configparser
>>> cfg = configparser.ConfigParser()
>>> cfg.read('settings.cfg')
['settings.cfg']
>>> cfg
<configparser.ConfigParser object at 0x1006be4d0>
>>> cfg['french']
<Section: french>
>>> cfg['french']['greeting']
'Bonjour'
>>> cfg['files']['bin']
'/usr/local/bin'
```

Other options are available, including fancier interpolation. See the `configparser` documentation. If you need deeper nesting than two levels, try YAML or JSON.

Binary Files

Some file formats were designed to store particular data structures but are neither relational nor NoSQL databases. The sections that follow present some of them.

Padded Binary Files and Memory Mapping

Padded binary files are similar to padded text files, but the contents may be binary, and the padding byte may be `\x00` instead of a space character. Each record has a fixed size, as does each field within a record. This makes it simpler to `seek()` throughout the file for the desired records and fields. Every operation on the data is manual, so this approach tends to be used only in very low-level (e.g., close to the hardware) situations.

Data in this form can be *memory mapped* with the standard mmap library. See some examples and the standard documentation.

Spreadsheets

Spreadsheets, notably Microsoft Excel, are widespread binary data formats. If you can save your spreadsheet to a CSV file, you can read it by using the standard `csv` module described earlier. This will work for a binary XLS file, xlrd, or Tablib (mentioned in "Tablib" on page 424).

HDF5

HDF5 is a binary data format for multidimensional or hierarchical numeric data. It's used mainly in science, where fast random access to large datasets (gigabytes to terabytes) is a common requirement. Even though HDF5 could be a good alternative to databases in some cases, for some reason HDF5 is almost unknown in the business world. It's best suited to *write once, read many* (*WORM*) applications for which database protection against conflicting writes is not needed. Here are some modules that you might find useful:

- `h5py` is a full-featured low-level interface. Read the documentation and code.
- `PyTables` is a bit higher-level, with database-like features. Read the documentation and code.

I'm mentioning HDF5 here in case you have a need to store and retrieve large amounts of data and are willing to consider something outside the box as well as the usual database solutions. A good example of this kind of data is the Million Song dataset, which has downloadable song data in HDF5 and SQLite formats.

TileDB

A recent successor to HDF5 for dense or spare array storage is TileDB. Install the Python interface (which includes the TileDB library itself) by running `pip install tiledb`. This is aimed at scientific data and applications.

Relational Databases

Relational databases are only about 40 years old but are ubiquitous in the computing world. You'll almost certainly have to deal with them at one time or another. When you do, you'll appreciate what they provide:

- Access to data by multiple simultaneous users
- Protection from corruption by those users
- Efficient methods to store and retrieve the data
- Data defined by *schemas* and limited by *constraints*
- *Joins* to find relationships across diverse types of data
- A declarative (rather than imperative) query language: Structured Query Language (SQL)

These are called *relational* because they show relationships among different kinds of data in the form of rectangular tables. For instance, in our menu example earlier, there is a relationship between each item and its price.

A *table* is a rectangular grid of *columns* (data fields) and *rows* (individual data records), similar to a spreadsheet. The intersection of a row and column is a table *cell*. To create a table, name it and specify the order, names, and types of its columns. Each row has the same columns, although a column may be defined to allow missing data (called *nulls*) in cells. In the menu example, you could create a table with one row for each item being sold. Each item has the same columns, including one for the price.

A column or group of columns is usually the table's *primary key*; its values must be unique in the table. This prevents adding the same data to the table more than once. This key is *indexed* for fast lookups during queries. An index works a little like a book index, making it fast to find a particular row.

Each table lives within a parent *database*, like a file within a directory. Two levels of hierarchy help keep things organized a little better.

Yes, the word *database* is used in multiple ways: as the server, the table container, and the data stored therein. If you'll be referring to all of them at the same time, it might help to call them *database server*, *database*, and *data*.

If you want to find rows by a nonkey column value, define a *secondary index* on that column. Otherwise, the database server must perform a *table scan*—a brute-force search of every row for matching column values.

Tables can be related to each other with *foreign keys*, and column values can be constrained to these keys.

SQL

SQL is not an API or a protocol, but a declarative *language*: you say *what* you want rather than *how* to do it. It's the universal language of relational databases. SQL queries are text strings sent by a client to the database server, which in turn figures out what to do with them.

There have been various SQL standard definitions, and all database vendors have added their own tweaks and extensions, resulting in many SQL *dialects*. If you store your data in a relational database, SQL gives you some portability. Still, dialect and operational differences can make it difficult to move your data to another type of database. SQL statements are divided into two main categories:

Data definition language (DDL)
: Handles creation, deletion, constraints, and permissions for tables, databases, and users

Data manipulation language (DML)
: Handles data insertions, selects, updates, and deletions

Table 23-1 lists the basic SQL DDL commands.

Table 23-1. Basic SQL DDL commands

Operation	SQL pattern	SQL example
Create a database	`CREATE DATABASE` *dbname*	`CREATE DATABASE d`
Select current database	`USE` *dbname*	`USE d`
Delete a database and its tables	`DROP DATABASE` *dbname*	`DROP DATABASE d`
Create a table	`CREATE TABLE` *tbname* `(` *coldefs* `)`	`CREATE TABLE t (id INT, count INT)`
Delete a table	`DROP TABLE` *tbname*	`DROP TABLE t`
Remove all rows from a table	`TRUNCATE TABLE` *tbname*	`TRUNCATE TABLE t`

Why all the CAPITAL LETTERS? SQL is case-insensitive, but it's a tradition (don't ask me why) to SHOUT its keywords in code examples to distinguish them from column names.

The main DML operations of a relational database are often known by the acronym CRUD:

- *C*reate by using the SQL `INSERT` statement
- *R*ead by using `SELECT`
- *U*pdate by using `UPDATE`
- *D*elete by using `DELETE`

Table 23-2 looks at the commands available for SQL DML.

Table 23-2. Basic SQL DML commands

Operation	SQL pattern	SQL example
Add a row	`INSERT INTO` *tbname* `VALUES( ... )`	`INSERT INTO t VALUES(7, 40)`
Select all rows and columns	`SELECT * FROM` *tbname*	`SELECT * FROM t`
Select all rows, some columns	`SELECT` *cols* `FROM` *tbname*	`SELECT id, count FROM t`

Operation	SQL pattern	SQL example
Select some rows, some columns	`SELECT` *cols* `FROM` *tbname* `WHERE` *condition*	`SELECT id, count from t WHERE count > 5 AND id = 9`
Change some rows in a column	`UPDATE` *tbname* `SET` *col* = *value* `WHERE` *condition*	`UPDATE t SET count=3 WHERE id=5`
Delete some rows	`DELETE FROM` *tbname* `WHERE` *condition*	`DELETE FROM t WHERE count <= 10 OR id = 16`

DB-API

An *application programming interface* (API) is a set of functions that you can call to get access to a service. DB-API is Python's standard API for accessing relational databases. Using it, you can write a single program that works with multiple kinds of relational databases instead of writing a separate program for each one. It's similar to Java Database Connectivity (JDBC) or Perl DBI.

The API's main functions are the following:

`connect()`
: Make a connection to the database; this can include arguments such as username, password, server address, and others.

`cursor()`
: Create a `cursor` object to manage queries.

`execute()` *and* `executemany()`
: Run one or more SQL commands against the database.

`fetchone()`, `fetchmany()`, *and* `fetchall()`
: Get the results from `execute()`.

The Python database modules in the coming sections conform to DB-API, often with extensions and some differences in details.

SQLite

SQLite is a good, light, open source relational database. It's implemented as a standard Python library and stores databases in normal files. These files are portable across machines and operating systems, making SQLite a very portable solution for simple relational database applications. It isn't as full featured as MySQL or PostgreSQL, but it does support SQL, and it manages multiple simultaneous users. Web browsers, smartphones, and other applications use SQLite as an embedded database.

You begin with a `connect()` to the local SQLite database file that you want to use or create. This file is the equivalent of the directory-like *database* that parents tables in other servers. The special string `:memory:` creates the database in memory only; this

is fast and useful for testing but will lose data when your program terminates or if your computer goes down.

For the next example, let's make a database called `enterprise.db` and the table `zoo` to manage our thriving roadside petting zoo business. The table columns are as follows:

`critter`
: A variable-length string and our primary key

`count`
: An integer count of our current inventory for this animal

`damages`
: The dollar amount of our current losses from animal–human interactions

Let's create the `zoo` table:

```
>>> import sqlite3
>>> conn = sqlite3.connect('enterprise.db')
>>> curs = conn.cursor()
>>> curs.execute('''CREATE TABLE zoo
    (critter VARCHAR(20) PRIMARY KEY,
     count INT,
     damages FLOAT)''')
<sqlite3.Cursor object at 0x1006a22d0>
```

Python's triple quotes are handy when creating long strings such as SQL queries.

Now, add some animals to the zoo:

```
>>> curs.execute('INSERT INTO zoo VALUES("duck", 5, 0.0)')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.execute('INSERT INTO zoo VALUES("bear", 2, 1000.0)')
<sqlite3.Cursor object at 0x1006a22d0>
```

You can use a safer way to insert data, using a *placeholder*:

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES(?, ?, ?)'
>>> curs.execute(ins, ('weasel', 1, 2000.0))
<sqlite3.Cursor object at 0x1006a22d0>
```

This time, we use three question marks in the SQL to indicate that we plan to insert three values, and then pass those three values as a tuple to the `execute()` function. Placeholders handle tedious details such as quoting. They protect you against *SQL injection*—a kind of external attack that inserts malicious SQL commands into the system (and which is common on the web).

Now, let's see if we can get all our animals out again:

```
>>> curs.execute('SELECT * FROM zoo')
<sqlite3.Cursor object at 0x1006a22d0>
>>> rows = curs.fetchall()
```

```
>>> print(rows)
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

Let's get them again, but ordered by their counts:

```
>>> curs.execute('SELECT * from zoo ORDER BY count')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0), ('bear', 2, 1000.0), ('duck', 5, 0.0)]
```

Hey, we wanted them in descending order:

```
>>> curs.execute('SELECT * from zoo ORDER BY count DESC')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

Which type of animal is costing us the most?

```
>>> curs.execute('''SELECT * FROM zoo WHERE
...     damages = (SELECT MAX(damages) FROM zoo)''')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0)]
```

You would have thought it was the bears. It's always best to check the actual data.

Before we leave SQLite, we need to clean up. If we opened a connection and a cursor, we need to close them when we're done:

```
>>> curs.close()
>>> conn.close()
```

DuckDB

DuckDB is similar to SQLite: it runs as part of your program, not a separate process or server. One difference is that its focus is online analytical processing (OLAP). DuckDB has SQL that's extended to read and write external file formats like CSV, JSON, and even Amazon Simple Storage Service (S3) buckets. I'll say more about DuckDB in Chapter 25.

MySQL

MySQL is a popular open source relational database. Unlike SQLite, it's an actual server, so clients can access it from different devices across the network.

Table 23-3 lists the drivers you can use to access MySQL from Python. For more details on all Python MySQL drivers, see the *python.org* wiki.

Table 23-3. MySQL drivers

Name	Link	Pypi package	Import as	Notes
MySQLdb	*https://https://mysqlclient.readthedocs.io*	mysql-connector-python	`MySQLdb`	
MySQL Connector	*https://bit.ly/mysql-cpdg*	mysql-connector-python	`mysql.connector`	
PyMySQL	*https://github.com/petehunt/PyMySQL*	pymysql	`pymysql`	
oursql	*https://pythonhosted.org/oursql*	oursql	`oursql`	Requires the MySQL C client libraries

PostgreSQL

PostgreSQL is a full-featured open source relational database. Indeed, in many ways, it's more advanced than MySQL. Table 23-4 presents the Python drivers you can use to access it.

Table 23-4. PostgreSQL drivers

Name	Link	PyPI package	Import as	Notes
Psycopg2	*https://www.psycopg.org*	psycopg2	psycopg2	Needs `pg_config` from PostgreSQL client tools
py-postgresql	*https://pypi.org/project/py-postgresql*	py-postgresql	postgresql	

The most popular driver is `psycopg2`, but its installation requires the PostgreSQL client libraries.

SQLAlchemy

SQL is not quite the same for all relational databases, and DB-API takes you only so far. Each database implements a particular *dialect* reflecting its features and philosophy. Many libraries try to bridge these differences in one way or another. The most popular cross-database Python library is SQLAlchemy.

It isn't in the standard library, but it's well known and used by many people. You can install it on your system by using this command:

```
$ pip install sqlalchemy
```

You can use SQLAlchemy on several levels:

- The lowest level manages open database connection *pools* (caching these saves startup time), executes SQL commands, and returns results. This is closest to the DB-API.

- Next up is the *SQL expression language*, which lets you express queries in a more Python-oriented way.
- Highest is the ORM (Object Relational Mapper) layer, which uses the SQL Expression Language and binds application code with relational data structures.

As we go along, you'll understand what the terms mean in those levels. SQLAlchemy works with the database drivers documented in the previous sections. You don't need to import the driver; the initial connection string you provide to SQLAlchemy will determine it. That string looks like this:

```
dialect + driver :// user : password @ host : port / dbname
```

The values you put in this string are as follows:

`dialect`
: The database type

`driver`
: The particular driver you want to use for that database

`user` and `password`
: Your database authentication strings

`host` and `port`
: The database server's location (`: port` is needed only if it's not the standard one for this server)

`dbname`
: The database to initially connect to on the server

Table 23-5 lists the dialects and drivers.

Table 23-5. SQLAlchemy connection

Dialect	Driver
`sqlite`	`pysqlite` (or omit)
`mysql`	`mysqlconnector`
`mysql`	`pymysql`
`mysql`	`oursql`
`postgresql`	`psycopg2`
`postgresql`	`pypostgresql`

See also the SQLAlchemy details on dialects for MySQL, SQLite, PostgreSQL, and other databases.

The engine layer

First, let's try the lowest level of SQLAlchemy, which does little more than the base DB-API functions.

Let's try it with SQLite, which is already built into Python. The connection string for SQLite skips the *`host`*, *`port`*, *`user`*, and *`password`*. The *`dbname`* tells SQLite which file to use to store your database. If you omit the *`dbname`*, SQLite builds a database in memory. If the *`dbname`* starts with a slash (/), it's an absolute filename on your computer (as in Linux and macOS; for example, *C:* on Windows). Otherwise, it's relative to your current directory.

The following segments are all part of one program, separated here for explanation.

To begin, you need to import what we need. The following is an example of an *import alias*, which lets us use the string `sa` to refer to SQLAlchemy methods. I do this mainly because `sa` is a lot easier to type than `sqlalchemy`:

```
>>> import sqlalchemy as sa
```

Connect to the database and create the storage for it in memory (the argument string `sqlite:///:memory:` also works):

```
>>> conn = sa.create_engine('sqlite://')
```

Create a database table called `zoo` that comprises three columns:

```
>>> conn.execute('''CREATE TABLE zoo
...     (critter VARCHAR(20) PRIMARY KEY,
...      count INT,
...      damages FLOAT)''')
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb10>
```

Running `conn.execute()` returns a SQLAlchemy object called a `ResultProxy`. You'll soon see what to do with it.

Now, insert three sets of data into your new empty table:

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES (?, ?, ?)'
>>> conn.execute(ins, 'duck', 10, 0.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb50>
>>> conn.execute(ins, 'bear', 2, 1000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef090>
>>> conn.execute(ins, 'weasel', 1, 2000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef450>
```

Next, ask the database for everything that we just put in:

```
>>> rows = conn.execute('SELECT * FROM zoo')
```

In SQLAlchemy, `rows` is not a list; it's that special `ResultProxy` object that we can't print directly:

```
>>> print(rows)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef9d0>
```

However, you can iterate over it like a list, so we can get a row at a time:

```
>>> for row in rows:
...     print(row)
...
('duck', 10, 0.0)
('bear', 2, 1000.0)
('weasel', 1, 2000.0)
```

That was almost the same as the SQLite DB-API example that you saw earlier. The one advantage is that we didn't need to import the database driver at the top; SQLAlchemy figured that out from the connection string. Just changing the connection string would make this code portable to another type of database. SQLAlchemy supports various SQL dialects. Another plus is SQLAlchemy's *connection pooling*, which you can read about at its documentation site.

The SQL Expression Language

The next level up is SQLAlchemy's SQL Expression Language. It introduces functions to create the SQL for various operations. The Expression Language handles more of the SQL dialect differences than the lower-level engine layer does. It can be a handy middle-ground approach for relational database applications.

Here's how to create and populate the `zoo` table. Again, these are successive fragments of a single program.

The import and connection are the same as before:

```
>>> import sqlalchemy as sa
>>> conn = sa.create_engine('sqlite://')
```

To define the `zoo` table, we begin using some of the Expression Language instead of SQL:

```
>>> meta = sa.MetaData()
>>> zoo = sa.Table('zoo', meta,
...     sa.Column('critter', sa.String, primary_key=True),
...     sa.Column('count', sa.Integer),
...     sa.Column('damages', sa.Float)
...    )
>>> meta.create_all(conn)
```

Check out the parentheses in that multiline call in the preceding example. The structure of the `Table()` method matches the structure of the table. Just as our table contains three columns, there are three calls to `Column()` inside the parentheses of the `Table()` method call.

Meanwhile, zoo is a magic object that bridges the SQL database world and the Python data structure world.

Insert the data with more Expression Language functions:

```
>>> conn.execute(zoo.insert(('bear', 2, 1000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017ea910>
>>> conn.execute(zoo.insert(('weasel', 1, 2000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eab10>
>>> conn.execute(zoo.insert(('duck', 10, 0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eac50>
```

Next, create the SELECT statement (`zoo.select()` selects everything from the table represented by the zoo object, just as `SELECT * FROM zoo` would do in plain SQL):

```
>>> result = conn.execute(zoo.select())
```

Finally, get the results:

```
>>> rows = result.fetchall()
>>> print(rows)
[('bear', 2, 1000.0), ('weasel', 1, 2000.0), ('duck', 10, 0.0)]
```

The Object-Relational Mapper

In the previous section, the zoo object was a midlevel connection between SQL and Python. At the top layer of SQLAlchemy, the ORM uses the SQL Expression Language but tries to make the actual database mechanisms invisible. You define classes, and the ORM handles how to get their data in and out of the database. The basic idea behind that complicated term, *object-relational mapper*, is that you can refer to objects in your code and thus stay close to the way Python likes to operate while still using a relational database.

We'll define a Zoo class and hook it into the ORM. This time, we make SQLite use the *zoo.db* file so that we can confirm that the ORM worked.

As in the previous two sections, the snippets that follow are actually one program separated by explanations. Don't worry if you don't understand some of it. The SQLAlchemy documentation has all the details—and this stuff can get complex. I just want you to get an idea of the amount of work this requires, so you can decide which of the approaches discussed in this chapter suits you.

The initial import is the same, but this time we need something else also:

```
>>> import sqlalchemy as sa
>>> from sqlalchemy.ext.declarative import declarative_base
```

Here, we make the connection:

```
>>> conn = sa.create_engine('sqlite:///zoo.db')
```

Now, we get into SQLAlchemy's ORM. We define the `Zoo` class and associate its attributes with table columns:

```
>>> Base = declarative_base()
>>> class Zoo(Base):
...     __tablename__ = 'zoo'
...     critter = sa.Column('critter', sa.String, primary_key=True)
...     count = sa.Column('count', sa.Integer)
...     damages = sa.Column('damages', sa.Float)
...     def __init__(self, critter, count, damages):
...         self.critter = critter
...         self.count = count
...         self.damages = damages
...     def __repr__(self):
...         return "<Zoo({}, {}, {})>".format(self.critter, self.count,
...           self.damages)
```

The following line magically creates the database and table:

```
>>> Base.metadata.create_all(conn)
```

You can then insert data by creating Python objects. The ORM manages these internally:

```
>>> first = Zoo('duck', 10, 0.0)
>>> second = Zoo('bear', 2, 1000.0)
>>> third = Zoo('weasel', 1, 2000.0)
>>> first
<Zoo(duck, 10, 0.0)>
```

Next, we get the ORM to take us to SQL land. We create a session to talk to the database:

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=conn)
>>> session = Session()
```

Within the session, we write the three objects that we created to the database. The `add()` function adds one object, and `add_all()` adds a list:

```
>>> session.add(first)
>>> session.add_all([second, third])
```

Finally, we need to force everything to complete:

```
>>> session.commit()
```

Did it work? Well, it created a *zoo.db* file in the current directory. You can use the command-line `sqlite3` program to check it:

```
$ sqlite3 zoo.db
SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
```

```
zoo
sqlite> select * from zoo;
duck|10|0.0
bear|2|1000.0
weasel|1|2000.0
```

The purpose of this section is to show what an ORM is and how it works at a high level. There is a full tutorial on the SQLAlchemy site. After reading this, decide which of the following levels would best fit your needs:

- Plain DB-API, as in "SQLite" on page 429
- The SQLAlchemy engine
- The SQLAlchemy Expression Language
- The SQLAlchemy ORM

Using an ORM to avoid the complexities of SQL seems like a natural choice. Should you use one? Some people think ORMs should be avoided, but others think the criticism is overdone. Whoever's right, an ORM is an abstraction, and all abstractions are leaky and break down at some point. When your ORM doesn't do what you want, you must figure out both how it works and how to fix it in SQL. To borrow an internet meme:

> Some people, when confronted with a problem, think, "I know, I'll use an ORM." Now they have two problems.

Use ORMs for simple applications or applications that map data pretty directly to database tables. If the application is that simple, you may consider using straight SQL or the SQL Expression Language.

Other Database Access Packages

If you're looking for Python tools that will handle multiple databases, with more features than the bare DB-API but less than SQLAlchemy, these are worth a look:

- dataset claims the goal "databases for lazy people." It's built on SQLAlchemy and provides a simple ORM for SQL, JSON, and CSV storage.
- Records bills itself as "SQL for humans." It supports only SQL queries, using SQLAlchemy internally to handle SQL dialect issues, connection pooling, and other details. Its integration with Tablib (mentioned in "Tablib" on page 424) lets you export data to CSV, JSON, and other formats.

NoSQL Data Stores

Relational tables can be visualized as rectangles (rows and columns), but data comes in many shapes and may be difficult to fit without significant effort and contortion. It's a square-peg-in-a-round-hole problem.

Some nonrelational databases have been written to allow more flexible data definitions as well as to process very large datasets or support custom data operations. These datasets have been collectively labeled *NoSQL* (formerly meaning *no SQL*, now the less confrontational *not only SQL*).

The simplest type of NoSQL databases are *key-value stores*. One popularity ranking from db-engines.com shows some that I cover in the following sections.

The dbm Family

The `dbm` formats were around long before the *NoSQL* label was coined. They're simple key-value stores, often embedded in applications such as web browsers to maintain various settings. A `dbm` database is like a Python dictionary in the following ways:

- You can assign a value to a key, and it's automatically saved to the database on disk.
- You can query a key for its value.

The following is a quick example. The second argument to the following `open()` method is `r` to read, `w` to write, and `c` for both, creating the file if it doesn't exist:

```
>>> import dbm
>>> db = dbm.open('definitions', 'c')
```

To create key-value pairs, just assign a value to a key just as you would for a dictionary:

```
>>> db['mustard'] = 'yellow'
>>> db['ketchup'] = 'red'
>>> db['pesto'] = 'green'
```

Let's pause and check what we have so far:

```
>>> len(db)
3
>>> db['pesto']
b'green'
```

Now close, then reopen, to see whether the program saved what we gave it:

```
>>> db.close()
>>> db = dbm.open('definitions', 'r')
>>> db['mustard']
b'yellow'
```

Keys and values are stored as `bytes`. You cannot iterate over the database object `db`, but you can get the number of keys by using `len()`. The `get()` and `setdefault()` functions work as they do for dictionaries.

Memcached

Memcached is a fast in-memory key-value *cache* server. It's often put in front of a database or used to store web-server session data.

You can download versions for Linux and macOS. If you want to try out this section, you'll need a running Memcached server and Python driver.

Many Python drivers exist; one that works with Python 3 is `python3-memcached`, which you can install by using this command:

```
$ pip install python-memcached
```

To use it, connect to a Memcached server, after which you can do the following:

- Set and get values for keys
- Increment or decrement a value
- Delete a key

Data keys and values are *not* persistent, and data that you wrote earlier might disappear. This is inherent in Memcached—it's a cache server, not a database, and it avoids running out of memory by discarding old data.

You can connect to multiple Memcached servers at the same time. In this next example, we're just talking to one on the same computer:

```
>>> import memcache
>>> db = memcache.Client(['127.0.0.1:11211'])
>>> db.set('marco', 'polo')
True
>>> db.get('marco')
'polo'
>>> db.set('ducks', 0)
True
>>> db.get('ducks')
0
>>> db.incr('ducks', 2)
2
>>> db.get('ducks')
2
```

Redis

Redis is a *data structure server*. It handles keys and their values, but the values are richer than those in other key-value stores. Like Memcached, all the data in a Redis server should fit in memory. Unlike Memcached, Redis can do the following:

- Save data to disk for reliability and restarts
- Keep old data
- Provide more data structures than simple strings

The Redis data types are a close match to Python's, and a Redis server can be a useful intermediary for one or more Python applications to share data. I've found it so useful that it's worth a little extra coverage here.

The Python driver `redis-py` has its source code and tests on GitHub as well as documentation. You can install it by using this command:

```
$ pip install redis
```

The Redis server has good documentation. If you install and start the Redis server on your local computer (with the network nickname `localhost`), you can try the programs in the following sections.

Strings

A key with a single value is a Redis *string*. Simple Python data types are automatically converted. Connect to a Redis server at a host (the default is `localhost`) and port (default is `6379`):

```
>>> import redis
>>> conn = redis.Redis()
```

Connecting to `redis.Redis('localhost')` or `redis.Redis('localhost', 6379)` would have given the same result.

List all keys (we have none so far):

```
>>> conn.keys('*')
[]
```

Set a simple string (key `secret`), integer (key `carats`), and float (key `fever`):

```
>>> conn.set('secret', 'ni!')
True
>>> conn.set('carats', 24)
True
>>> conn.set('fever', '101.5')
True
```

Get the values back (as Python `byte` values) by key:

```
>>> conn.get('secret')
b'ni!'
>>> conn.get('carats')
b'24'
>>> conn.get('fever')
b'101.5'
```

Here, the `setnx()` method sets a value only if the key does not exist:

```
>>> conn.setnx('secret', 'icky-icky-icky-ptang-zoop-boing!')
False
```

It failed because we had already defined `secret`:

```
>>> conn.get('secret')
b'ni!'
```

The `getset()` method returns the old value and sets it to a new one at the same time:

```
>>> conn.getset('secret', 'icky-icky-icky-ptang-zoop-boing!')
b'ni!'
```

Let's not get too far ahead of ourselves. Did it work?

```
>>> conn.get('secret')
b'icky-icky-icky-ptang-zoop-boing!'
```

Now, get a substring by using `getrange()` (as in Python, offset `0` means start, and `-1` means end):

```
>>> conn.getrange('secret', -6, -1)
b'boing!'
```

Replace a substring by using `setrange()` (using a zero-based offset):

```
>>> conn.setrange('secret', 0, 'ICKY')
32
>>> conn.get('secret')
b'ICKY-icky-icky-ptang-zoop-boing!'
```

Next, set multiple keys at once by using `mset()`:

```
>>> conn.mset({'pie': 'cherry', 'cordial': 'sherry'})
True
```

Get more than one value at once by using `mget()`:

```
>>> conn.mget(['fever', 'carats'])
[b'101.5', b'24']
```

Delete a key by using `delete()`:

```
>>> conn.delete('fever')
True
```

Increment by using the `incr()` or `incrbyfloat()` commands, and decrement with `decr()`:

```
>>> conn.incr('carats')
25
>>> conn.incr('carats', 10)
35
>>> conn.decr('carats')
34
>>> conn.decr('carats', 15)
19
>>> conn.set('fever', '101.5')
True
>>> conn.incrbyfloat('fever')
102.5
>>> conn.incrbyfloat('fever', 0.5)
103.0
```

There's no `decrbyfloat()`. Use a negative increment to reduce the fever:

```
>>> conn.incrbyfloat('fever', -2.0)
101.0
```

Lists

Redis lists can contain only strings. The list is created when you do your first insertion. Insert at the beginning by using `lpush()`:

```
>>> conn.lpush('zoo', 'bear')
1
```

Insert more than one item at the beginning:

```
>>> conn.lpush('zoo', 'alligator', 'duck')
3
```

Insert before or after a value by using `linsert()`:

```
>>> conn.linsert('zoo', 'before', 'bear', 'beaver')
4
>>> conn.linsert('zoo', 'after', 'bear', 'cassowary')
5
```

Insert at an offset by using `lset()` (the list must exist already):

```
>>> conn.lset('zoo', 2, 'marmoset')
True
```

Insert at the end by using `rpush()`:

```
>>> conn.rpush('zoo', 'yak')
6
```

Get the value at an offset by using `lindex()`:

```
>>> conn.lindex('zoo', 3)
b'bear'
```

Get the values in an offset range by using `lrange()` (`0` to `-1` for all):

```
>>> conn.lrange('zoo', 0, 2)
[b'duck', b'alligator', b'marmoset']
```

Trim the list with `ltrim()`, keeping only those in a range of offsets:

```
>>> conn.ltrim('zoo', 1, 4)
True
```

Get a range of values (use `0` to `-1` for all) by using `lrange()`:

```
>>> conn.lrange('zoo', 0, -1)
[b'alligator', b'marmoset', b'bear', b'cassowary']
```

Hashes

Redis *hashes* are similar to Python dictionaries but can contain only strings. Also, you can go only one level deep, not make deep-nested structures. Here are examples that create and play with a Redis hash called `song`:

Set the fields `do` and `re` in hash `song` at once by using `hmset()`:

```
>>> conn.hmset('song', {'do': 'a deer', 're': 'about a deer'})
True
```

Set a single field value in a hash by using `hset()`:

```
>>> conn.hset('song', 'mi', 'a note to follow re')
1
```

Get one field's value by using `hget()`:

```
>>> conn.hget('song', 'mi')
b'a note to follow re'
```

Get multiple field values by using `hmget()`:

```
>>> conn.hmget('song', 're', 'do')
[b'about a deer', b'a deer']
```

Get all field keys for the hash by using `hkeys()`:

```
>>> conn.hkeys('song')
[b'do', b're', b'mi']
```

Get all field values for the hash by using `hvals()`:

```
>>> conn.hvals('song')
[b'a deer', b'about a deer', b'a note to follow re']
```

Get the number of fields in the hash by using `hlen()`:

```
>>> conn.hlen('song')
3
```

Get all field keys and values in the hash by using `hgetall()`:

```
>>> conn.hgetall('song')
{b'do': b'a deer', b're': b'about a deer', b'mi': b'a note to follow re'}
```

Set a field if its key doesn't exist by using `hsetnx()`:

```
>>> conn.hsetnx('song', 'fa', 'a note that rhymes with la')
1
```

Sets

Redis sets are similar to Python sets, as you'll see in the following examples.

Add one or more values to a set:

```
>>> conn.sadd('zoo', 'duck', 'goat', 'turkey')
3
```

Get the number of values from the set:

```
>>> conn.scard('zoo')
3
```

Get all the set's values:

```
>>> conn.smembers('zoo')
{b'duck', b'goat', b'turkey'}
```

Remove a value from the set:

```
>>> conn.srem('zoo', 'turkey')
True
```

Let's make a second set to show a set operations:

```
>>> conn.sadd('better_zoo', 'tiger', 'wolf', 'duck')
0
```

Intersect (get the common members of) the `zoo` and `better_zoo` sets:

```
>>> conn.sinter('zoo', 'better_zoo')
{b'duck'}
```

Get the intersection of `zoo` and `better_zoo`, and store the result in the set `fowl_zoo`:

```
>>> conn.sinterstore('fowl_zoo', 'zoo', 'better_zoo')
1
```

Who's in there?

```
>>> conn.smembers('fowl_zoo')
{b'duck'}
```

Get the union (all members) of `zoo` and `better_zoo`:

```
>>> conn.sunion('zoo', 'better_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

Store that union result in the set `fabulous_zoo`:

```
>>> conn.sunionstore('fabulous_zoo', 'zoo', 'better_zoo')
4
>>> conn.smembers('fabulous_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

What does `zoo` have that `better_zoo` doesn't? Use `sdiff()` to get the set difference, and `sdiffstore()` to save it in the `zoo_sale` set:

```
>>> conn.sdiff('zoo', 'better_zoo')
{b'goat'}
>>> conn.sdiffstore('zoo_sale', 'zoo', 'better_zoo')
1
>>> conn.smembers('zoo_sale')
{b'goat'}
```

Sorted sets

One of the most versatile Redis data types is the *sorted set*, or *zset*. It's a set of unique values, but each value has an associated floating-point *score*. You can access each item by its value or score. Sorted sets have many uses:

- Leaderboards
- Secondary indices
- Time series, using timestamps as scores

We show the last use case, tracking user logins via timestamps. We're using the Unix *epoch* value that's returned by the Python `time()` function:

```
>>> import time
>>> now = time.time()
>>> now
1361857057.576483
```

Let's add our first guest, looking nervous:

```
>>> conn.zadd('logins', 'smeagol', now)
1
```

Five minutes later, another guest:

```
>>> conn.zadd('logins', 'sauron', now+(5*60))
1
```

Two hours later:

```
>>> conn.zadd('logins', 'bilbo', now+(2*60*60))
1
```

One day later, not hasty:

```
>>> conn.zadd('logins', 'treebeard', now+(24*60*60))
1
```

In what order did `bilbo` arrive?

```
>>> conn.zrank('logins', 'bilbo')
2
```

When was that?

```
>>> conn.zscore('logins', 'bilbo')
1361864257.576483
```

Let's see everyone in login order:

```
>>> conn.zrange('logins', 0, -1)
[b'smeagol', b'sauron', b'bilbo', b'treebeard']
```

With their times, please:

```
>>> conn.zrange('logins', 0, -1, withscores=True)
[(b'smeagol', 1361857057.576483), (b'sauron', 1361857357.576483),
(b'bilbo', 1361864257.576483), (b'treebeard', 1361943457.576483)]
```

Caches and expiration

All Redis keys have a time to live, or *expiration date*. By default, this is forever. We can use the `expire()` function to instruct Redis how long to keep the key. The value is a number of seconds:

```
>>> import time
>>> key = 'now you see it'
>>> conn.set(key, 'but not for long')
True
>>> conn.expire(key, 5)
True
>>> conn.ttl(key)
5
>>> conn.get(key)
b'but not for long'
>>> time.sleep(6)
>>> conn.get(key)
>>>
```

The `expireat()` command expires a key at a given epoch time. Key expiration is useful to keep caches fresh and to limit login sessions. An analogy: in the refrigerated room behind the milk racks at your grocery store, store employees yank those gallons as they reach their freshness dates.

Redis Alternative: Valkey?

After all that detail on Redis, I should mention some nontechnical news that might make you look for an alternative. In 2024, the Redis license was changed from open source. There are commercial reasons: cloud providers could charge for services based on open source tools, but none of the money flowed to the tool developers.

After this license change, many alternative tools popped up. The best choice now seems to be Valkey, which has these characteristics:

- API-compatible with Redis
- Maintained by experienced Redis core developers
- As performant as Redis, or better

See the Valkey documentation for the Valkey server installation steps. Then use `pip install valkey` to get the Python driver. Read the valkey-py docs for details and walk-throughs of the Python API.

Document Databases

A *document database* is a NoSQL database that stores data with varying fields. Compared to a relational table (with evenly shaped rows and columns), such data is "ragged," with varying fields (columns) per row and even nested fields. You could handle data like this in memory with Python dictionaries and lists, or store it as JSON files. To store such data in a relational database table, you would need to define every possible column and use nulls for missing data.

ODM can stand for *object data manager* or *object document mapper* (at least they agree on the *O* part). An ODM is the document database counterpart of a relational database ORM. Some popular document databases and tools (drivers and ODMs) are listed in Table 23-6.

Table 23-6. Document databases

Database	Python API
Mongo	PyMongo
DynamoDB	Boto3
CouchDB	couchdb

PostgreSQL can do some of the things that document databases do. Some of its extensions allow it to escape relational orthodoxy while keeping features like transactions, data validation, and foreign keys: multidimensional *arrays* store more than one value in a table cell, and *jsonb* stores JSON data in a cell, with full indexing and querying.

Time-Series Databases

Time-series data may be collected at fixed intervals (such as computer performance metrics) or at random times, which has led to many storage methods. Among many of these, some with Python support are listed in Table 23-7.

Table 23-7. Temporal databases

Database	Python API
InfluxDB	influx-client
kdb+	PyQ
Prometheus	prometheus-client
TigerData	PostgreSQL clients
OpenTSDB	potsdb
PyStore	PyStore

Graph Databases

For our last case of data that needs its own database category, we have *graphs*: *nodes* (data) connected by *edges* or *vertices* (relationships). An individual X *user* could be a node, with edges to other users like *following* and *followed*.

Graph data has become more visible with the growth of social media, where the value is in the connections as much as the content. Some popular graph databases are outlined in Table 23-8.

Table 23-8. Graph databases

Database	Python API
Neo4j	neo4j
OrientDB	pyorient
ArangoDB	pyArango

Other NoSQL

The NoSQL servers listed here handle data larger than memory, and many of them use multiple computers. Table 23-9 presents notable servers and their Python libraries.

Table 23-9. NoSQL databases

Database	Python API
Apache Cassandra	cassandra-driver
CouchDB	couchdb-python
Apache HBase	HappyBase
Kyoto Cabinet	kyotocabinet
MongoDB	PyMongo
Riak	riak

Full-Text Databases

Finally, a special category of databases is for *full-text* search. These databases index everything, so you can find that poem that talks about windmills and giant wheels of cheese. You can see some popular open source examples along with their Python APIs in Table 23-10.

Table 23-10. Full-text databases

Databases	Python API
Apache Lucene	PyLucene
Apache Solr	pysolr
Elasticsearch	elasticsearch
Sphinx	sphinxapi-py3
Xapian	xapian
Whoosh	whoosh

Vector Databases

The recent kaboom of AI has led to ways to access the internal data of machine learning: arrays (vectors) of numbers that fill the hidden layers of machine learning systems. These are useful for similarity searching, rather than the exact matching used in relational databases. The following are some vector databases that have Python access:

- Chroma
- VectorDB
- pgvector

Geospatial Databases

Looking through our database warehouse, even more are on the shelf. These are extensions to existing relational databases that have been mentioned earlier:

- PostGIS
- GeoPandas
- SpatiaLite

Review/Preview

This chapter sang of databases, housing all types of data for all uses.

Next: the web. It's hard to believe that the web has been around for only a few decades and how much it has influenced our lives.

Practice

23.1 Save the following text lines to a file called *books.csv* (notice that if the fields are separated by commas, you need to surround any comma-containing field with quotes):

```
author,book
J R R Tolkien,The Hobbit
Lynne Truss,"Eats, Shoots & Leaves"
```

23.2 Use the `csv` module and its `DictReader` method to read *books.csv* to the variable `books`. Print the values in `books`. Did `DictReader` handle the quotes and commas in the second book's title?

23.3 Create a CSV file called *books2.csv* by using these lines:

```
title,author,year
The Weirdstone of Brisingamen,Alan Garner,1960
Perdido Street Station,China Miéville,2000
Thud!,Terry Pratchett,2005
The Spellman Files,Lisa Lutz,2007
Small Gods,Terry Pratchett,1992
```

23.4 Use the `sqlite3` module to create a SQLite database called *books.db* and a table called `books` with these fields: `title` (text), `author` (text), and `year` (integer).

23.5 Read *books2.csv* and insert its data into the `book` table.

23.6 Select and print the `title` column from the `book` table in alphabetical order.

23.7 Select and print all columns from the `book` table in order of publication.

23.8 Use the SQLAlchemy module to connect to the sqlite3 database *books.db* that you just made in Exercise 23.4. As in 23.6, select and print the `title` column from the `book` table in alphabetical order.

23.9 Install the Redis server and the Python redis library (`pip install redis`) on your computer. Create a Redis hash called `test` with the fields `count` (`1`) and `name` (`Fester Bestertester`). Print all the fields for `test`.

23.10 Increment the `count` field of `test` and print it.

CHAPTER 24
The Web

Oh, what a tangled web we weave...
—Walter Scott, "Marmion"

Straddling the French–Swiss border is the European Organization for Nuclear Research (CERN)—a particle physics research institute that smashes atoms multiple times, just to make sure. All that smashing generates a mountain of data. In 1989, the English scientist Tim Berners-Lee first circulated a proposal within CERN to help disseminate information there and across the research community. He called it the *World Wide Web* and distilled its design into three simple ideas:

Hypertext Transfer Protocol (HTTP)
: A protocol for web clients and servers to interchange requests and responses

Hypertext Markup Language (HTML)
: A presentation format for results

Uniform resource locator (URL)
: A way to uniquely represent a server and a *resource* on that server

In its simplest usage, a web client (Berners-Lee was the first to use the term *browser*) connected to a web server with HTTP, requested a URL, and received HTML. This was all built on the networking base from the *internet*, which at the time was non-commercial, and known only to a few universities and research organizations.

He wrote the first web browser and server on a NeXT computer.[1] Web awareness really expanded in 1993, when a group of students at the University of Illinois

1 NeXT was a company founded by Steve Jobs during his exile from Apple.

released the Mosaic[2] web browser (for Windows, the Macintosh, and Unix) and the National Center for Supercomputing Applications (NCSA) *httpd* server.

When I downloaded Mosaic that summer and started building sites, I had no idea that the web and the internet would soon become part of everyday life. The internet was still officially noncommercial then; there were about five hundred known web servers in the world.

By the end of 1994, the number of web servers had grown to 10,000. The internet was opened to commercial use, and the authors of Mosaic founded Netscape to write commercial web software. Netscape went public as part of the early internet frenzy, and the web's explosive growth has never stopped.

Almost every computer language has been used to write web clients and web servers. The dynamic languages Perl, PHP, and Ruby have been especially popular. In this chapter, I show why Python is a particularly good language for web work at every level:

- Clients, to access remote sites
- Servers, to provide data for websites and web APIs
- Web APIs and services, to interchange data in ways other than viewable web pages

Web Basics

The low-level network plumbing of the internet is called Transmission Control Protocol/Internet Protocol, or more commonly, simply TCP/IP (see "TCP/IP" on page 381). It moves bytes among computers but doesn't care what those bytes mean. That's the job of higher-level *protocols*—syntax definitions for specific purposes. HTTP is the standard protocol for web data interchange.

The web is a client-server system. The client makes a *request* to a server: the client opens a TCP/IP connection, sends the URL and other information via HTTP, and receives a *response*.

The format of the response is also defined by HTTP. It includes the status of the request, and (if the request succeeded) the response's data and format.

The most well-known web client is a web *browser*. It can make HTTP requests in multiple ways. You might manually initiate a request by typing a URL into the

2 One of its authors, Marc Andreesen, used the blue color for web links, added the IMG tag to HTML, joined Netscape, and is now a billionaire venture capitalist.

location bar or clicking a link in a web page. Often, the data returned is used to display a website (HTML documents, JavaScript files, CSS files, and images), but it can be any type of data, not just that intended for display.

An important aspect of HTTP is that it's *stateless*. Each HTTP connection that you make is independent of all the others. This simplifies basic web operations but complicates others. Here are just a few samples of the challenges:

Caching
: Remote content that doesn't change should be saved by the web client and used to avoid downloading from the server again.

Sessions
: A shopping website should remember the contents of your shopping cart.

Authentication
: Sites that require your username and password should remember them while you're logged in.

Solutions to statelessness include *cookies*, which are small pieces of specific information sent by the server to the client so that the server can then identify the client as unique when the client sends the cookie back.

HTTP Testing

To start, we'll try different ways of connecting to websites with utility programs. This helps show how HTTP works, before we dig into Python web clients and servers.

Telnet

HTTP is a text-based protocol, so you can type it yourself for web testing. The ancient Telnet program lets you connect to any server and port, and type commands to any service that's running there. For secure (encrypted) connections to other machines, it's been replaced by Secure Shell (SSH).

Let's ask everyone's favorite test site, Google, some basic information about its home page. Type this:

```
$ telnet www.google.com 80
```

If a web server is on port 80 (this is where unencrypted `http` usually runs; encrypted `https` uses port 443) at *google.com* (I think that's a safe bet), `telnet` will print some reassuring information, and then display a final blank line that's your cue to type something else:

```
Trying 74.125.225.177...
Connected to www.google.com.
Escape character is '^]'.
```

Now, type an actual HTTP command for Telnet to send to the Google web server. The most common HTTP command (the one your browser uses when you type a URL in its location bar) is `GET`. This retrieves the contents of the specified resource, such as an HTML file, and returns it to the client. For our first test, we'll use the HTTP command `HEAD`, which retrieves basic information *about* the resource:

```
HEAD / HTTP/1.1
```

Add an extra carriage return to send a blank line so the remote server knows you're all done and want a response. That `HEAD /` sends the HTTP `HEAD` *verb* (command) to get information about the home page (`/`). You'll receive a response such as Example 24-1. (I replaced some of the long lines with … so they wouldn't stick out of the book.)

Example 24-1. Results from an HTTP GET query to www.google.com

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
Content-Security-Policy-Report-Only: ...
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Date: Mon, 20 Jan 2025 12:48:30 GMT
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Transfer-Encoding: chunked
Expires: Mon, 20 Jan 2025 12:48:30 GMT
Cache-Control: private
Set-Cookie: AEC=...
Set-Cookie: NID=...
```

These are HTTP response headers and their values. Some, like `Date` and `Content-Type`, are required. Others, such as `Set-Cookie`, are used to track your activity across multiple visits (we talk about *state management* a little later in this chapter). When you make an HTTP `HEAD` request, you get back only headers. If you had used the HTTP `GET` or `POST` commands, you would also receive data from the home page (a mixture of HTML, CSS, JavaScript, and whatever else Google decided to throw into its home page).

I don't want to leave you stranded in Telnet. To close it, type `q`.

curl

Using Telnet is simple but is a completely manual process. The curl program is probably the most popular command-line web client. Documentation includes the book *Everything Curl* by Daniel Stenberg. A table compares curl with similar tools. The download page includes all the major platforms and many obscure ones.

The simplest use of curl does an implicit `GET` (output truncated here):

```
$ curl http://www.example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    ...
```

This uses `HEAD`:

```
$ curl --head http://www.example.com
HTTP/1.1 200 OK
Content-Type: text/html
ETag: "84238dfc8092e5d9c0dac8ef93371a07:1736799080.121134"
Last-Modified: Mon, 13 Jan 2025 20:11:20 GMT
Cache-Control: max-age=420
Date: Mon, 20 Jan 2025 12:53:07 GMT
Connection: keep-alive
```

If you're passing arguments, you can include them in the command line or a data file. In these examples, I use the following:

- *`url`* for any website
- `data.txt` as a text data file with these contents: `a=1&b=2`
- `data.json` as a JSON data file with these contents: `{"a":1, "b": 2}`
- `a=1&b=2` as two data arguments

Using default (*form-encoded*) arguments:

```
$ curl -X POST -d "a=1&b=2" url
$ curl -X POST -d "@data.txt" url
```

For JSON-encoded arguments:

```
$ curl -X POST -d "{'a':1,'b':2}" -H "Content-Type: application/json" url
$ curl -X POST -d "@data.json" url
```

HTTPie

A more Pythonic alternative to curl is HTTPie:

```
$ pip install httpie
```

To make a form-encoded POST, similar to the methods for curl shown previously (`-f` is a synonym for `--form`), use the following:

```
$ http -f POST url a=1 b=2
$ http POST -f url < data.txt
```

The default encoding is JSON:

```
$ http POST url a=1 b=2
$ http POST url < data.json
```

HTTPie also handles HTTP headers, cookies, file uploads, authentication, redirects, SSL, and so on. As usual, see the docs.

httpbin

You can test your web queries against the site *httpbin*, or download and run the site in a local Docker image:

```
$ docker run -p 80:80 kennethreitz/httpbin
```

Web Clients

The previous section showed various utilities connecting with web servers. Now let's look at how to write web clients in Python.

The Standard Library

In Python 2, web client and server modules were a bit scattered. One of the Python 3 goals was to bundle these modules into two *packages* (remember from Chapter 12 that a package is just a directory containing module files):

- `http` manages all the client-server HTTP details:
  - `client` does the client-side stuff.
  - `server` helps you write Python web servers.
  - `cookies` and `cookiejar` manage cookies, which save data between site visits.
- `urllib` runs on top of `http`:
  - `request` handles the client request.
  - `response` handles the server response.
  - `parse` cracks the parts of a URL.

If you're trying to write code that's compatible with both Python 2 and Python 3, keep in mind that `urllib` changed a lot between the two versions. For a better alternative, refer to "Requests" on page 460.

Let's use the standard library to get something from a website. The URL in the following example returns information from a test website:

```
>>> import urllib.request as ur
>>>
>>> url = 'http://www.example.com/'
>>> conn = ur.urlopen(url)
```

This little chunk of Python opens a TCP/IP connection to the remote web server *www.example.com*, makes an HTTP request, and receives an HTTP response. The response contains more than just the page data. In the official documentation, we find that `conn` is an `HTTPResponse` object with multiple methods and attributes. One of the most important parts of the response is the HTTP *status code*:

```
>>> print(conn.status)
200
```

A `200` means that everything is peachy. Dozens of HTTP status codes exist, grouped into five ranges by their first (hundreds) digit:

1xx (information)
: The server received the request but has extra information for the client.

2xx (success)
: It worked; every success code other than 200 conveys extra details.

3xx (redirection)
: The resource moved, so the response returns the new URL to the client.

4xx (client error)
: A problem has occurred from the client side, such as the well-known 404 (not found). The 418 code (*I'm a teapot*) was an April Fools' joke.

5xx (server error)
: 500 is the generic whoops; you might see a 502 (bad gateway) if a disconnect between a web server and a backend application server exists.

To get the actual data contents from the web page, use the `read()` method of the `conn` variable. This returns a `bytes` value. Let's get the data and print the first 50 bytes:

```
>>> data = conn.read()
>>> print(data[:50])

b'<!doctype html>\n<html>\n<head>\n    <title>Example D'
```

We can convert these bytes to a string and print its first 50 characters:

```
>>> str_data = data.decode('utf8')
>>> print(str_data[:50])
<!doctype html>
<html>
<head>
    <title>Example D
>>>
```

The rest is more HTML and CSS.

Out of sheer curiosity, which HTTP headers were sent back to us?

```
>>> for key, value in conn.getheaders():
...     print(key, value)
...
Cache-Control max-age=604800
Content-Type text/html; charset=UTF-8
Date Sun, 05 May 2019 03:09:26 GMT
Etag "1541025663+ident"
Expires Sun, 12 May 2019 03:09:26 GMT
Last-Modified Fri, 09 Aug 2013 23:54:35 GMT
Server ECS (agb/5296)
Vary Accept-Encoding
X-Cache HIT
Content-Length 1270
Connection close
```

Remember that Telnet example a little earlier? Now, our Python library is parsing all those HTTP response headers and providing them in a dictionary. `Date` and `Server` seem straightforward; some of the others, less so. It's helpful to know that HTTP has a set of standard headers, such as `Content-Type`, and many optional ones.

Requests

For most purposes, I think web client development with Requests is easier than using the standard Python equivalents. I'll show the basics of Requests in this section and use this package throughout this book for web client tasks.

First, install the Requests library with `pip install requests`.

Let's redo our previous *example.com* query with Requests:

```
>>> import requests
>>> resp = requests.get('http://example.com')
>>> resp
<Response [200]>
>>> resp.status_code
200
>>> resp.text[:50]
'<!doctype html>\n<html>\n<head>\n    <title>Example D'
```

Now let's try rewriting some code that you saw way back in Example 1-5. It accessed a Wayback Machine API, using the standard libraries urllib.request and json. Example 24-2 shows the original version again.

Example 24-2. ia1.py

```
import webbrowser
import json
from urllib.request import urlopen

print("Let's find an old website.")
site = input("Type a website URL: ")
era = input("Type a year, month, and day, like 20150613: ") + "0000"
url = f"http://archive.org/wayback/available?url={site}&timestamp={era}"
response = urlopen(url)
contents = response.read()
text = contents.decode("utf-8")
data = json.loads(text)
try:
    old_site = data["archived_snapshots"]["closest"]["url"]
    print("Found this copy:", old_site)
    print("It should appear in your browser now.")
    webbrowser.open(old_site)
except:
    print("Sorry, no luck finding", site)
```

Now let's redo that example by using Requests in Example 24-3.

Example 24-3. ia2.py

```
import webbrowser
import requests

print("Let's find an old website.")
site = input("Type a website URL: ")
era = input("Type a year, month, and day, like 20250714: ") + "0000"
url = f"http://archive.org/wayback/available?url={site}&timestamp={era}"
response = requests.get(url)
data = response.json()
try:
    old_site = data["archived_snapshots"]["closest"]["url"]
    print("Found this copy:", old_site)
    print("It should appear in your browser now.")
    webbrowser.open(old_site)
except:
    print("Sorry, no luck finding", site)
```

This saves a few steps and seems a bit more natural. Browse the documentation (which is pretty good) for full details.

Other Web Clients

AIOHTTP is a purely async alternative to Requests and can be used to build servers as well as clients. Install with `pip install aiohttp`.

HTTPX does what requests does, and more—especially, async HTTP connections. Install it with `pip install httpx`.

A recent comparison by Georges Haidar recommends the following:

- Simple synchronous calls: Requests
- Both synchronous and asynchronous calls: HTTPX
- Asynchronous calls only: AIOHTTP

A good example for async is web scraping. If you're accessing multiple websites, most of the time you're waiting for the following:

- Client request → server (I/O)
- Server (processing your request; now where did I put that file again?)
- Server results → client (I/O)

For this use case, async can be orders of magnitude faster. I talked about async in Chapter 21, but here's a comparison of Requests and HTTPX accessing 10 websites. Example 24-4 compares the run times of the following:

- An empty function
- Sequential Requests calls
- Requests with a single session
- Async HTTPX

Example 24-4. bench.py

```
import asyncio
import time
import requests
import httpx

urls = [
"https://example.com/tech-seo-vercel-gcp/z",
"https://example.com/firebase-django-ruby/S",
"https://example.com/docker-css-css/E",
"https://example.com/git-mysql-sitemap/q",
"https://example.com/performance-cdn-conversion-rate/4",
"https://example.com/azure-joomla-mvc/A",
```

```
"https://example.com/performance-sitemap-aws/U",
"https://example.com/drupal-server-bounce-rate/H",
"https://example.com/scrum-prototype-laravel/K",
"https://example.com/frontend-performance-azure/8",
]

def timer(func):
    def inner(*args, **kwargs):
        t1 = time.perf_counter()
        func()
        t2 = time.perf_counter()
        print(f"{func.__name__:30} {(t2 - t1):.2f}")
    return inner

@timer
def run_nothing():
    pass

@timer
def run_requests():
    results = [requests.get(url) for url in urls]

@timer
def run_requests_session():
    s = requests.session()
    results = [s.get(url) for url in urls]

async def httpx_async():
    async with httpx.AsyncClient() as client:
        tasks = [asyncio.create_task(client.get(url)) for url in urls]
        await asyncio.gather(*tasks)

@timer
def run_httpx_async():
    asyncio.run(httpx_async())


if __name__ == "__main__":
    run_nothing()
    run_requests()
    run_requests_session()
    run_httpx_async()
```

I ran the code twice. The second time is faster from system caching. You can see that running requests with a session is faster than individual calls, and that async HTTPX is even faster:

```
$ python bench.py
run_nothing                    0.00
run_requests                   6.24
run_requests_session           3.10
run_httpx_async                0.89
```

```
$ python bench.py
run_nothing                    0.00
run_requests                   1.32
run_requests_session           0.50
run_httpx_async                0.25
```

Web Servers

Web developers have found Python to be an excellent language for writing web servers and server-side programs. This has led to so many Python-based web *frameworks* that it can be hard to navigate among them and make choices—not to mention deciding which deserve to go into a book.

A web framework provides features that you can use to build websites, so it does more than a simple web (HTTP) server. You'll see features such as routing (URL to server function), templates (HTML with dynamic inclusions), debugging, and more.

I'm not going to cover all the frameworks here, just those that I've found to be relatively simple to use and suitable for real websites.

The Simplest Python Web Server

You can run a simple web server by typing just one line of Python:

```
$ python -m http.server
```

This implements a bare-bones Python HTTP server. If no problems occur, this will print an initial status message:

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

That `0.0.0.0` means *any TCP address*, so web clients can access it no matter what address the server has. More low-level details on TCP and other network plumbing are presented in Chapter 22.

You can now request files, with paths relative to your current directory, and they will be returned. If you type *http://localhost:8000* in your web browser, you should see a directory listing there, and the server will print access log lines such as this:

```
127.0.0.1 - - [20/Feb/2025 22:02:37] "GET / HTTP/1.1" 200 -
```

`localhost` and `127.0.0.1` are TCP synonyms for *your local computer*, so this works regardless of whether you're connected to the internet. You can interpret this line as follows:

- `127.0.0.1` is the client's IP address.
- The first `-` is the remote username, if found.
- The second `-` is the login username, if required.
- `[20/Feb/2025 22:02:37]` is the access date and time.
- `GET / HTTP/1.1` is the command sent to the web server:
  - The HTTP method (`GET`)
  - The resource requested (`/`, the top)
  - The HTTP version (`HTTP/1.1`)
- The final `200` is the HTTP status code returned by the web server.

Click any file. If your browser can recognize the format (HTML, PNG, GIF, JPEG, and so on), the browser should display the file, and the server will log the request. For instance, if you have the file *oreilly.png* in your current directory, a request for *http://localhost:8000/oreilly.png* should return the image of the unsettling fellow in Figure 24-1.

Figure 24-1. The O'Reilly tarsier returns

The log should show something such as this:

```
127.0.0.1 - - [25/Feb/2025 22:03:48] "GET /oreilly.png HTTP/1.1" 200 -
```

If you have other files in the same directory on your computer, they should show up in a listing on your display, and you can click any one to download it. If your browser is configured to display that file's format, you'll see the results on your screen; otherwise, your browser will ask if you want to download and save the file.

The default port number used is 8000, but you can specify another:

```
$ python -m http.server 9999
```

You should see this:

```
Serving HTTP on 0.0.0.0 port 9999 ...
```

This Python-only server is best suited for quick tests. You can stop it by killing its process; in most terminals, press Ctrl+C.

You should not use this basic server for a busy production website. Traditional web servers such as those by Apache and NGINX are much faster for serving static files. In addition, this simple server has no way to handle dynamic content, which more extensive servers can do by accepting parameters.

Web Server Gateway Interface

All too soon, the allure of serving simple files wears off, and we want a web server that can also run programs dynamically. In the early days of the web, the *Common Gateway Interface* (*CGI*) was designed for clients to make web servers run external programs and return the results. CGI also handled getting input arguments from the client through the server to the external programs. However, the programs were started anew for *each* client access. This could not scale well, because even small programs have appreciable startup time.

To avoid this startup delay, people began merging the language interpreter into the web server. Apache ran PHP within its `mod_php` module, Perl in `mod_perl`, and Python in `mod_python`. Then, code in these dynamic languages could be executed within the long-running Apache process itself rather than in external programs.

An alternative method was to run the dynamic language within a separate long-running program and have it communicate with the web server. FastCGI and SCGI are examples.

Python web development made a leap with the definition of the *Web Server Gateway Interface* (*WSGI*), a universal API between Python web applications and web servers. All the Python web frameworks and web servers in the rest of this chapter use WSGI. You don't normally need to know how WSGI works (there really isn't much to it), but it helps to know what some of the parts under the hood are called. This is a *synchronous* connection—one step follows another.

ASGI

In a few places so far, I've mentioned that Python has been introducing *asynchronous* language features like `async`, `await`, and `asyncio`. Asynchronous Server Gateway Interface (ASGI) is a counterpart of WSGI that uses these new features. For more discussion and examples of new frameworks that use ASGI, see "asyncio" on page 369.

Apache

The Apache HTTP Server's best WSGI module is `mod_wsgi`. This can run Python code within the Apache process or in separate processes that communicate with Apache.

You should already have Apache if your system is Linux or macOS. For Windows, you'll need to install Apache.

Finally, install your preferred WSGI-based Python web framework. Let's try Bottle here. Almost all the work involves configuring Apache, which can be a dark art.

Create the test file shown in Example 24-5 and save it as *home.wsgi*.

Example 24-5. home.wsgi

```
import bottle

application = bottle.default_app()

@bottle.route('/')
def home():
    return "apache and wsgi, sitting in a tree"
```

Do not call `run()` this time, because that starts the built-in Python web server. We need to assign a name to the variable `application` because that's what `mod_wsgi` looks for to marry the web server and the Python code.

If Apache and its `mod_wsgi` module are working correctly, we just need to connect them to our Python script. We want to add one line to the file that defines the default website for this Apache server, but finding that file is a task itself. It could be */etc/apache2/httpd.conf*, or */etc/apache2/sites-available/default*, or the Latin name of someone's pet salamander.

Let's assume for now that you understand Apache and found that file. Add this line inside the `<VirtualHost>` section that governs the default website:

```
        WSGIScriptAlias / /var/www/test/home.wsgi
```

That section might then look like this:

```
    <VirtualHost *:80>
        DocumentRoot /var/www

        WSGIScriptAlias / /var/www/test/home.wsgi

        <Directory /var/www/test>
        Order allow,deny
        Allow from all
```

```
    </Directory>
</VirtualHost>
```

Start `apache`, or restart it if it was running to make it use this new configuration. If you then browse to *http://localhost/*, you should see this:

```
apache and wsgi, sitting in a tree
```

This runs `mod_wsgi` in *embedded mode*, as part of Apache itself.

You can also run it in *daemon mode*, as one or more processes, separate from Apache. To do this, add two new directive lines to your `apache` config file:

```
WSGIDaemonProcess domain-name user=user-name group=group-name threads=25
WSGIProcessGroup domain-name
```

In the preceding example, *`user-name`* and *`group-name`* are the operating system user and group names, and the *`domain-name`* is the name of your internet domain. A minimal Apache config might look like this:

```
<VirtualHost *:80>
    DocumentRoot /var/www

    WSGIScriptAlias / /var/www/test/home.wsgi

    WSGIDaemonProcess mydomain.com user=myuser group=mygroup threads=25
    WSGIProcessGroup mydomain.com

    <Directory /var/www/test>
    Order allow,deny
    Allow from all
    </Directory>
</VirtualHost>
```

NGINX

The `nginx` web server does not have an embedded Python module. Instead, it's a frontend to a separate WSGI server such as uWSGI or gUnicorn. Together they make a fast and configurable platform for Python web development.

Other Python Web Servers

Following are some of the independent Python-based WSGI servers that work like `apache` or `nginx`, using multiple processes and/or threads (see Chapter 23) to handle simultaneous requests:

- CherryPy
- Pylons
- Waitress

Here are some *event-based* servers, which use a single process but avoid blocking on any single request:

- Tornado
- gevent
- Gunicorn

I have more to say about events in the discussion about *concurrency* in Chapter 21.

Web Server Frameworks

Web servers handle the HTTP and WSGI details, but you use web *frameworks* to write the Python code that powers the site. So, let's talk about frameworks for a while and then get back to alternative ways of serving sites that use them.

If you want to write a website in Python, there are many (some say too many) Python web frameworks. A web framework handles, at a minimum, client requests and server responses. Most major web frameworks include these tasks:

- Handle HTTP
- Perform authentication (*authn*, or *who are you*?)
- Perform authorization (*authz*, or *what can you do*?)
- Manage sessions
- Get parameters
- Validate parameters (required/optional, type, range)
- Handle HTTP verbs
- Route (functions/classes)
- Serve static files (HTML, JS, CSS, images)
- Serve dynamic data (databases, services)
- Return values and HTTP status

Optional features include the following:

- Backend templates
- Database connectivity, ORMs
- Rate limiting
- Asynchronous tasks

The coming sections contain example code for two frameworks (Bottle and Flask). These are *synchronous*. Later, I talk about alternatives, especially for database-backed websites. You can find a Python framework to power any site that you can think of.

Bottle

Bottle consists of a single Python file, so it's easy to try out, and it's easy to deploy later. Bottle isn't part of standard Python, so to install it, type **`pip install bottle`**.

Here's code that will run a test web server and return a line of text when your browser accesses the URL *http://localhost:9999/*. Save it as *bottle1.py* (Example 24-6).

Example 24-6. bottle1.py

```
from bottle import route, run

@route('/')
def home():
  return "It isn't fancy, but it's my home page"

run(host='localhost', port=9999)
```

Bottle uses the `route` decorator to associate a URL with the following function; in this case, `/` (the home page) is handled by the `home()` function. Make Python run this server script by typing this:

```
$ python bottle1.py
```

You should see this on your browser when you access *http://localhost:9999/*:

```
It isn't fancy, but it's my home page
```

The `run()` function executes Bottle's built-in Python test web server. You don't need to use this for Bottle programs, but it's useful for initial development and testing.

Now, instead of creating text for the home page in code, let's make a separate HTML file called *index.html* that contains this line of text:

```
My <b>new</b> and <i>improved</i> home page!!!
```

Make Bottle return the contents of this file when the home page is requested. Save this script as *bottle2.py* (Example 24-7).

Example 24-7. bottle2.py

```
from bottle import route, run, static_file

@route('/')
def main():
    return static_file('index.html', root='.')

run(host='localhost', port=9999)
```

In the call to `static_file()`, we want the file *index.html* in the directory indicated by `root` (in this case, `.`, the current directory). If your previous server example code was still running, stop it. Now, run the new server:

```
$ python bottle2.py
```

When you ask your browser to get *http:/localhost:9999/*, you should see this:

```
My new and improved home page!!!
```

Let's add one last example that shows how to pass arguments to a URL and use them. Of course, this will be *bottle3.py*, which you can see in Example 24-8.

Example 24-8. bottle3.py

```
from bottle import route, run, static_file

@route('/')
def home():
    return static_file('index.html', root='.')

@route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s!" % thing

run(host='localhost', port=9999)`
```

We have a new function called `echo()` and want to pass it a string argument in a URL. That's what the line `@route('/echo/<thing>')` in the preceding example does. That `<thing>` in the route means that whatever was in the URL after `/echo/` is assigned to the string argument `thing`, which is then passed to the `echo` function. To see what happens, stop the old server if it's still running and then start it with the new code:

```
$ python bottle3.py
```

Then access *http://localhost:9999/echo/Mothra* in your web browser. You should see the following:

```
Say hello to my little friend: Mothra!
```

Now, leave *bottle3.py* running for a minute so that we can try something else. You've been verifying that these examples work by typing URLs into your browser and looking at the displayed pages. You can also use client libraries such as Requests to do your work for you. Save this as *bottle_test.py* (Example 24-9).

Example 24-9. bottle_test.py

```
import requests

resp = requests.get('http://localhost:9999/echo/Mothra')
if resp.status_code == 200 and \
  resp.text == 'Say hello to my little friend: Mothra!':
    print('It worked! That almost never happens!')
else:
    print('Argh, got this:', resp.text)
```

Great! Now, run it:

```
$ python bottle_test.py
```

You should see this in your terminal:

```
It worked! That almost never happens!
```

This is a little example of a *unit test*. Chapter 15 provides more details on why tests are good and how to write them in Python.

There's more to Bottle than I've shown here. In particular, you can try adding these arguments when you call `run()`:

- `debug=True` creates a debugging page if you get an HTTP error.
- `reloader=True` reloads the page in the browser if you change any of the Python code.

Bottle is well documented at the developer site.

Flask

Bottle is a good initial web framework. If you need a few more cowbells and whistles, try Flask. It started in 2010 as an April Fools' joke, but enthusiastic response encouraged the author, Armin Ronacher, to make it a real framework. He named the result *Flask* as a wordplay on *Bottle*.

Flask is about as simple to use as Bottle, but it supports many extensions that are useful in professional web development, such as Facebook authentication and database integration. Flask balances ease of use with a rich feature set, but, like an Ikea bookshelf, some assembly is required.

The Flask package includes the Werkzeug WSGI library and the Jinja2 template library. You can install Flask with `pip install flask`.

Let's replicate the final Bottle example code in Flask. First, though, we need to make a few changes:

- Flask's default home directory for static files is *static*, and URLs for files there also begin with */static*. We change the folder to `.` (current directory) and the URL prefix to `''` (empty) to allow the URL `/` to map to the file *index.html*.
- In the `run()` function, setting `debug=True` also activates the automatic reloader; Bottle used separate arguments for debugging and reloading.

Save this file to *flask1.py* (Example 24-10).

Example 24-10. flask1.py

```
from flask import Flask

app = Flask(__name__, static_folder='.', static_url_path='')

@app.route('/')
def home():
    return app.send_static_file('index.html')

@app.route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s" % thing

app.run(port=9999, debug=True)
```

Then run the server from a terminal or window:

```
$ python flask1.py
```

Test the home page by typing this URL into your browser: *http://localhost:9999/*.

You should see the following (as you did for Bottle):

```
My new and improved home page!!!
```

Try the */echo* endpoint: *http://localhost:9999/echo/Godzilla*.

You should see this:

```
Say hello to my little friend: Godzilla
```

Setting `debug` to `True` when calling `run` has another benefit. If an exception occurs in the server code, Flask returns a specially formatted page with useful details about what went wrong, and where. Even better, you can type some commands to see the values of variables in the server program.

Do not set `debug = True` in production web servers. It exposes too much information about your server to potential intruders.

So far, the Flask example replicates what we did with Bottle. What can Flask do that Bottle can't? Flask includes Jinja2, a more extensive templating system. Here's a tiny example of how to use Jinja2 and Flask together.

Create a directory called *templates* and a file within it called *flask2.html* (Example 24-11).

Example 24-11. flask2.html

```
<html>
<head>
<title>Flask2 Example</title>
</head>
<body>
Say hello to my little friend: {{ thing }}
</body>
</html>
```

Next, we write the server code to grab this template, fill in the value of `thing` that we passed it, and render it as HTML (I'm dropping the `home()` function here to save space). Save this as *flask2.py* (Example 24-12).

Example 24-12. flask2.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/echo/<thing>')
def echo(thing):
    return render_template('flask2.html', thing=thing)

app.run(port=9999, debug=True)
```

That `thing = thing` argument means to pass a variable named `thing` to the template, with the value of the string `thing`.

Ensure that *flask1.py* isn't still running, and then start *flask2.py*:

```
$ python flask2.py
```

Now, type this URL: *http://localhost:9999/echo/Gamera*.

You should see the following:

```
Say hello to my little friend: Gamera
```

Let's modify our template and save it in the *templates* directory as *flask3.html*:

```
<html>
<head>
<title>Flask3 Example</title>
</head>
<body>
Say hello to my little friend: {{ thing }}.
Alas, it just destroyed {{ place }}!
</body>
</html>
```

You can pass this second argument to the `echo` URL in many ways. One approach is to pass an argument as part of the URL path.

Using this method, you simply extend the URL itself. Save the code shown in Example 24-13 as *flask3a.py*.

Example 24-13. flask3a.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/echo/<thing>/<place>')
def echo(thing, place):
    return render_template('flask3.html', thing=thing, place=place)

app.run(port=9999, debug=True)
```

As usual, stop the previous test server script if it's still running and then try this new one:

```
$ python flask3a.py
```

The URL would look like this: *http://localhost:9999/echo/Rodan/McKeesport*. And you should see the following:

```
Say hello to my little friend: Rodan. Alas, it just destroyed McKeesport!
```

Or you can provide the arguments as `GET` parameters, as shown in Example 24-14; save this as *flask3b.py*.

Example 24-14. flask3b.py

```
from flask import Flask, render_template, request

app = Flask(__name__)
```

```
@app.route('/echo/')
def echo():
    thing = request.args.get('thing')
    place = request.args.get('place')
    return render_template('flask3.html', thing=thing, place=place)

app.run(port=9999, debug=True)
```

Run the new server script:

```
$ python flask3b.py
```

This time, use this URL: *http://localhost:9999/echo?thing=Gorgo&place=Wilmerding*.

You should get back what you see here:

```
Say hello to my little friend: Gorgo. Alas, it just destroyed Wilmerding!
```

When a `GET` command is used for a URL, any arguments are passed in the form `&key1=val1&key2=val2&...`.

You can also use the dictionary `**` operator to pass multiple arguments to a template from a single dictionary (call this *flask3c.py*), as shown in Example 24-15.

Example 24-15. flask3c.py

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/echo/')
def echo():
    kwargs = {}
    kwargs['thing'] = request.args.get('thing')
    kwargs['place'] = request.args.get('place')
    return render_template('flask3.html', **kwargs)

app.run(port=9999, debug=True)
```

That `**kwargs` acts like `thing=thing, place=place`. It saves some typing if you have a lot of input arguments.

The Jinja2 templating language does a lot more than this. If you've programmed in PHP, you'll see many similarities.

Django

Django is a popular Python web framework, especially for large sites—even *very* large sites like Instagram. This framework is worth learning for many reasons, including frequent requests for Django experience in Python job ads. Django includes ORM

code (see "The Object-Relational Mapper" on page 436) to create automatic web pages for the typical database *CRUD* functions (create, replace, update, delete) that we looked at in Chapter 23. It also includes some automatic admin pages for these, but they're designed for internal use by programmers rather than public web page use. You don't have to use Django's ORM if you prefer another, such as SQLAlchemy, or direct SQL queries.

FastAPI

FastAPI handles both synchronous (WSGI) and asynchronous (ASGI) calls, uses type hints, generates test pages, and is extremely well documented. I recommend using it. I've written a separate book about FastAPI, but I'll condense the salient parts here:

Asynchronous
: FastAPI supports (asynchronous) ASGI, thanks to the Starlette library. But it also handles the traditional WSGI (synchronous) calls.

Routing
: The connection between URLs and functions is similar to Flask, using decorators.

Documentation
: World-class documentation and automatic test-form generation.

Performance
: Faster than most Python frameworks, largely from ASGI support.

Data validation
: Pydantic checks data types and values, greatly reducing many lines of traditional web verification code.

Litestar

Litestar is a newer framework. I have not yet worked with it in production, but it looks very good. It's similar to FastAPI:

- Synchronous and asynchronous (ASGI) workflows
- Uses type hints
- Self-documenting with OpenAPI
- Websocket support

It also goes beyond a bit, with extra support for the following:

- ORM for database
- Extension plug-ins
- Even better performance
- Authentication and authorization
- CORS and CSRF protection
- HTMX
- Stoplight as well as OpenAPI for API design
- Data validation methods beyond pydantic

Here's a curated list of Litestar links.

Database Frameworks

The web and databases are the peanut butter and jelly of computing: where you find one, you'll eventually find the other. In real-life Python applications, at some point you'll probably need to provide a web interface (site and/or API) to a relational database.

You could build your own with these:

- A web framework like Bottle or Flask
- A database package, like DB-API or SQLAlchemy
- A database driver, like PyMySQL

Instead, you could use a web/database package like one of these:

- Connexion
- Datasette
- sandman2
- Flask-Restless

Or you could use a framework with built-in database support, like Django.

Your database may not be a relational one. If your data schema varies significantly (with columns that differ markedly across rows), considering a *schemaless* database might be worthwhile, such as one of the NoSQL databases discussed in Chapter 23.

Web Services and Automation

We've just looked at traditional web client and server applications, consuming and generating HTML pages. Yet the web has turned out to be a powerful way to glue applications and data in many more formats than HTML.

webbrowser

Let's start with a little surprise. Start a Python session in a terminal window and type the following:

```
>>> import antigravity
```

This secretly calls the standard library's `webbrowser` module and directs your browser to an enlightening Python link.[3]

You can use this module directly. This program loads the main Python site's page in your browser:

```
>>> import webbrowser
>>> url = 'http://www.python.org/'
>>> webbrowser.open(url)
True
```

This opens the page in a new window:

```
>>> webbrowser.open_new(url)
True
```

And this opens the page in a new tab, if your browser supports tabs:

```
>>> webbrowser.open_new_tab('http://www.python.org/')
True
```

The `webbrowser` module makes your browser do all the work.

webview

Rather than calling your browser as `webbrowser` does, `webview` displays the page in its own window, using your machine's native GUI. To install on Linux or macOS, use `pip install pywebview[qt]`. For Windows: `pip install pywebview[cef]`. See the installation notes if you have problems.

3 If you don't see the link for some reason, visit xkcd.

Here's an example that gives `webview` the official US government current time site:

```
>>> import webview
>>> url = input("URL? ")
URL? http://time.gov
>>> webview.create_window(f"webview display of {url}", url)
```

To stop the program, kill the display window.

Web APIs and REST

Often data is available only within web pages. If you want to access it, you need to access the pages through a web browser and read it. If the authors of the website made any changes since the last time you visited, the location and style of the data might have changed.

Instead of publishing web pages, you can provide data through a web *application programming interface* (API). Clients access your service by making requests to URLs and getting back responses containing status and data. Instead of HTML pages, the data is in formats that are easier for programs to consume, such as JSON or XML (refer to Chapter 25 for more about these formats).

Representational State Transfer (*REST*) was defined by Roy Fielding in his doctoral thesis. Many products claim to have a *REST interface* or a *RESTful interface*. In practice, this often means only that they have a *web* interface—definitions of URLs to access a web service.

A *RESTful* service uses the HTTP *verbs* in specific ways:

- `HEAD` gets information about the resource but not its data.
- `GET` retrieves the resource's data from the server. This is the standard method used by your browser. `GET` should not be used to create, change, or delete data.
- `POST` creates a new resource.
- `PUT` replaces an existing resource, creating it if it doesn't exist.
- `PATCH` partially updates a resource.
- `DELETE` deletes. Truth in advertising!

A RESTful client can also request one or more content types from the server by using HTTP request headers. For example, a complex service with a REST interface might prefer its input and output to be JSON strings.

WebSockets

HTTP is stateless, so it requires a new connection between the client and server every time the client wants something. Oh, and the server can't initiate a connection with a client.

A whole category of applications is handled better by a continuous, two-way connection between a client and server. Here are some examples:

- Stock tickers
- Games
- Collaborative document editing
- Chat
- Notifications
- Asset tracking

This two-way connection is what WebSockets does. Example 24-16 shows a simple WebSockets server and client, copied from the site I just referenced. First, use `pip install websockets`.

Example 24-16. websocket_server.py

```
import asyncio

from websockets.asyncio.server import serve

async def hello(websocket):
    name = await websocket.recv()
    print(f"<<< {name}")

    greeting = f"Hello {name}!"

    await websocket.send(greeting)
    print(f">>> {greeting}")

async def main():
    async with serve(hello, "localhost", 8765):
        await asyncio.get_running_loop().create_future()  # run forever

if __name__ == "__main__":
    asyncio.run(main())
```

Run the server in the background, and its built-in interactive client:

```
$ python websocket_server.py &
```

Example 24-17 is a client that says hello to the server, gets its response, and prints it.

Example 24-17. websocket_client.py

```
import asyncio
from websockets.asyncio.client import connect

async def hello():
    async with connect("ws://localhost:8765") as websocket:
        await websocket.send("Hello world!")
        message = await websocket.recv()
        print(message)


if __name__ == "__main__":
    asyncio.run(hello())
```

Run the client:

```
$ python websocket_client.py
<<< Hello world!
>>> Hello Hello world!!
Hello Hello world!!
```

You can also try the server's built-in interactive client:

```
$ python -m websocket websocket_server.py
```

The web framework FastAPI (see "FastAPI" on page 477) includes WebSockets support, letting you mix HTTP and WebSocket endpoints.

Webhooks

I mentioned notifications in the previous section. That's also one of the use cases for webhooks. Rather than a URL, you pass a webhook name (a string) to a server that responds as an HTTP client.

FastAPI includes direct and well-documented support for webhooks.

Web Frontends

A web *frontend* directly faces the user and has the following characteristics:

- Is usually written in a combination of JavaScript, HTML, and CSS
- May communicate with a *backend* server, using tools like those already discussed in this chapter, via APIs or RPCs

You'll see in Chapter 25 that *data visualization* applications can be handled with all-in-one Python tools. In this section, we'll look at all-in-one Python packages that can produce traditional text-oriented web pages.

htmx

htmx is a JavaScript library that hides the JavaScript. Instead, you specify special HTML tags that run JavaScript invisibly. The library lets you do things that plain HTML can't:

- Any HTML element (not just `A` and `FORM`) can make HTTP requests to a backend.
- HTTP requests can be generated by other events (not just `A` clicks or `FORM` submits).
- Any HTML verb (not just `GET` and `POST`) can be used.
- AJAX requests can be made, with HTML results targeted to any HTML element.

This lets developers who aren't fluent in modern JavaScript create single- and multi-page applications in HTML. The htmx documentation has a useful page of examples.

FastHTML

A traditional web service might have a Python backend and generate web pages in various ways:

- Full frontend (e.g., React) components with JSON APIs to a backend
- A backend returning separate *static* HTML and JavaScript files
- A backend mixing code and HTML with Jinja2 *templates*

FastHTML is a newer Python library that can produce full web pages. A walk-through summarizes how it works and shows example code. Python functions like `Div()` and `P()` mirror their HTML (and our new friend htmx!) counterparts.

To install it, use `pip install python-fasthtml`. Example 24-18 is a minimal example, adapted from some of the documents pages.

Example 24-18. fast.py

```
from fasthtml.common import FastHTML, serve

app = FastHTML()

@app.get("/")
def home():
    return "<p>Yes, it works.</p>"

serve()
```

That's server code, similar to what you might create with Flask or FastAPI. If you saved that to the file *fast.py*, you run it in a test web server with `python fast.py`.

Give the URL *localhost:5001* to your web browser, and you'll see: `Yes, it works.` The test web server will print these logs:

```
INFO:     Will watch for changes in these directories: ['...']
INFO:     Uvicorn running on http://0.0.0.0:5001 (Press CTRL+C to quit)
INFO:     Started reloader process [46447] using WatchFiles
INFO:     Started server process [46449]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:49884 - "GET / HTTP/1.1" 200 OK
INFO:     127.0.0.1:49884 - "GET /favicon.ico HTTP/1.1" 404 Not Found
```

FastHTML might appeal to Python developers who, like me, can no longer claim to be "full stack" developers. Instead of editing an HTML file (with or without htmx features), you write everything in Python.

Crawl and Scrape

Sometimes you might want a little bit of information (for example, a movie rating, stock price, or product availability), but what you need is available only in HTML pages, surrounded by ads and extraneous content.

You could manually extract what you're looking for by doing the following:

1. Type the URL into your browser.
2. Wait for the remote page to load.
3. Look through the displayed page for the information you want.
4. Write it down somewhere.
5. Possibly repeat the process for related URLs.

However, automating some or all of these steps is much more satisfying. An automated web fetcher is called a *crawler* or *spider*.[4] After the contents have been retrieved from the remote web servers, a *scraper* parses it to find the needle in the haystack.

Scrapy

If you need an industrial-strength combined crawler *and* scraper, Scrapy is worth downloading. The `pip install scrapy` command installs the module and a standalone command-line Scrapy program.

4 Unappealing terms to arachnophobes.

Scrapy is a framework, not just a module such as Beautiful Soup. Scrapy does more, but it's more complex to set up. To learn more about Scrapy, read "Scrapy at a Glance" and the tutorial, both in the documentation.

Beautiful Soup

If you already have the HTML data from a website and just want to extract data from it, Beautiful Soup is a good choice. HTML parsing is harder than it sounds. This is because much of the HTML on public web pages is technically invalid: unclosed tags, incorrect nesting, and other complications. If you try to write your own HTML parser by using regular expressions (discussed in "Text Strings: Regular Expressions" on page 299), you'll soon encounter these messes.

To install Beautiful Soup, type the following command (don't forget the final `4`, or `pip` will try to install an older version and probably fail): `pip install beautifulsoup4`.

Now, let's use Beautiful Soup to get all the links from a web page. The HTML `a` element represents a link, and `href` is its attribute representing the link destination. Example 24-19 defines the function `get_links()` to do the grunt work, and a main program to get one or more URLs as command-line arguments.

Example 24-19. links.py

```
def get_links(url):
    import requests
    from bs4 import BeautifulSoup as soup
    result = requests.get(url)
    page = result.text
    doc = soup(page, features="html.parser")
    links = [element.get('href') for element in doc.find_all('a')]
    return links

if __name__ == '__main__':
    import sys
    for url in sys.argv[1:]:
        print('Links in', url)
        for num, link in enumerate(get_links(url), start=1):
            print(num, link)
        print()
```

I saved this program as *links.py* and then ran this command:

```
$ python links.py http://boingboing.net
```

Here are the first few lines that it printed:

```
Links in http://boingboing.net
1 https://boingboing.net
2 https://boingboing.net/search
```

```
3 https://store.boingboing.net
4 https://boingboing.net/search
5 https://store.boingboing.net
6 https://boingboing.net/blog
7 https://bbs.boingboing.net
8 https://bbs.boingboing.net/faq
9 https://store.boingboing.net
10 https://bit.ly/boingboingdealssupport
```

Requests-HTML

Kenneth Reitz, the author of the popular web client package Requests, has written a new scraping library called requests-html (for Python 3.6 and newer versions). It gets a page and processes its elements, so you can find, for example, all its links, or all the contents or attributes of any HTML element.

Requests-HTML has a clean design, similar to Requests and other packages by the same author. Overall, it may be easier to use than Beautiful Soup or Scrapy.

Let's Watch a Movie

Let's build a full program.

This program searches for videos by using an API at the Internet Archive.[5] This is one of the few APIs that allows anonymous access *and* should still be around after this book is printed.

Most web APIs require you to first get an *API key* and then provide it every time you access that API. Why? It's the tragedy of the commons: free resources with anonymous access are often overused or abused. That's why we can't have nice things.

The following program (save Example 24-20 as *iamovie.py*) does the following:

- Prompts you for part of a movie or video title
- Searches for it at the Internet Archive
- Returns a list of identifiers, names, and descriptions
- Lists them and asks you to select one
- Displays that video in your web browser

5 As you may remember, I used another Archive API in the main example program we looked at in Chapter 1.

Example 24-20. iamovie.py

```
"""Find a video at the Internet Archive
by a partial title match and display it."""

import sys
import webbrowser
import requests

def search(title):
    """Return a list of 3-item tuples (identifier,
       title, description) about videos
       whose titles partially match :title."""
    search_url = "https://archive.org/advancedsearch.php"
    params = {
        "q": f"title:({title}) AND mediatype:(movies)",
        "fl": "identifier,title,description",
        "output": "json",
        "rows": 20,
        "page": 1,
        }
    resp = requests.get(search_url, params=params)
    data = resp.json()
    docs = [(doc["identifier"], doc["title"], doc["description"])
            for doc in data["response"]["docs"]]
    return docs

def choose(docs):
    """Print line number, title and truncated description for
       each tuple in :docs. Get the user to pick a line
       number. If it's valid, return the first item in the
       chosen tuple (the "identifier"). Otherwise, return None."""
    last = len(docs) - 1
    for num, doc in enumerate(docs):
        print(f"{num}: ({doc[1]}) {doc[2][:30]}...")
    index = input(f"Which would you like to see (0 to {last})? ")
    try:
        return docs[int(index)][0]
    except:
        return None

def display(identifier):
    """Display the Archive video with :identifier in the browser"""
    details_url = f"https://archive.org/details/{identifier}"
    print("Loading", details_url)
    webbrowser.open(details_url)

def main(title):
    """Find any movies that match :title.
       Get the user's choice and display it in the browser."""
    identifiers = search(title)
    if identifiers:
```

```
        identifier = choose(identifiers)
        if identifier:
            display(identifier)
        else:
            print("Nothing selected")
    else:
        print("Nothing found for", title)

if __name__ == "__main__":
    main(sys.argv[1])
```

The `search()` function uses Requests to access the URL, get the results, and convert them to JSON. The other functions handle everything else. You'll see usage of list comprehensions, string slices, and other things that you've seen in previous chapters.

Here's what I got when I ran this program and searched for *eegah*:[6]

```
$ python iamovie.py eegah
0: (Cinema Insomnia presents Eegah) Used with permission for The V...
1: (Fright Night Theatre presents Eegah!) Used with permission for the V...
2: (Creature Feature S1-E15 - The Wasp Woman - Eegah) Air Date 04/10/2021 ...
3: (It's "Eegah" - Part 2) Wait till you see how this end...
4: (It's "Eegah" - Part 1) Count Gore De Vol shares some ...
5: (Eegah) This film has fallen into the ...
6: (Eegah) Teenagers stumble across a pre...
7: (eegah) Horror...
8: (EEGAH trailer) The infamous modern-day cavema...
9: (Midnight Movie show: eegah) Arch Hall Jr...
10: (Eegah) From IMDb : While driving thro...
11: (Eegah 1962 Slightly Enhanced) This is a slightly "enhanced" ...
12: (Eegah) A giant caveman has lived near...
13: (Bordello of Horror presents Eegah) The Vortexx Archives For use o...
14: (Hexan Arcane presents Eegah 1962) The Horror Host Network The Vo...
15: (Eegah ½ 1962 [1:32:32]; Comedy/Fantasy [Great Quality];) Often consid ...
16: (Eegah (1962)) While driving through the dese...
17: (Stoneham Cinema - Eegah) Eegah...
18: (EEGAH (1962) RE EDIT) A Re-edit of the notorious 196...
19: (SCCtv's Professor Fred's Movie Marvels - Eegah) SCCtv's Professor ...
Which would you like to see (0 to 19)? 18
Loading https://archive.org/details/eegah-1962-re-edit
```

The program displayed the page in my browser, ready to run. Figure 24-2 is a screenshot of the top.

6 With Richard Kiel as the caveman, years before he was Jaws in a Bond movie.

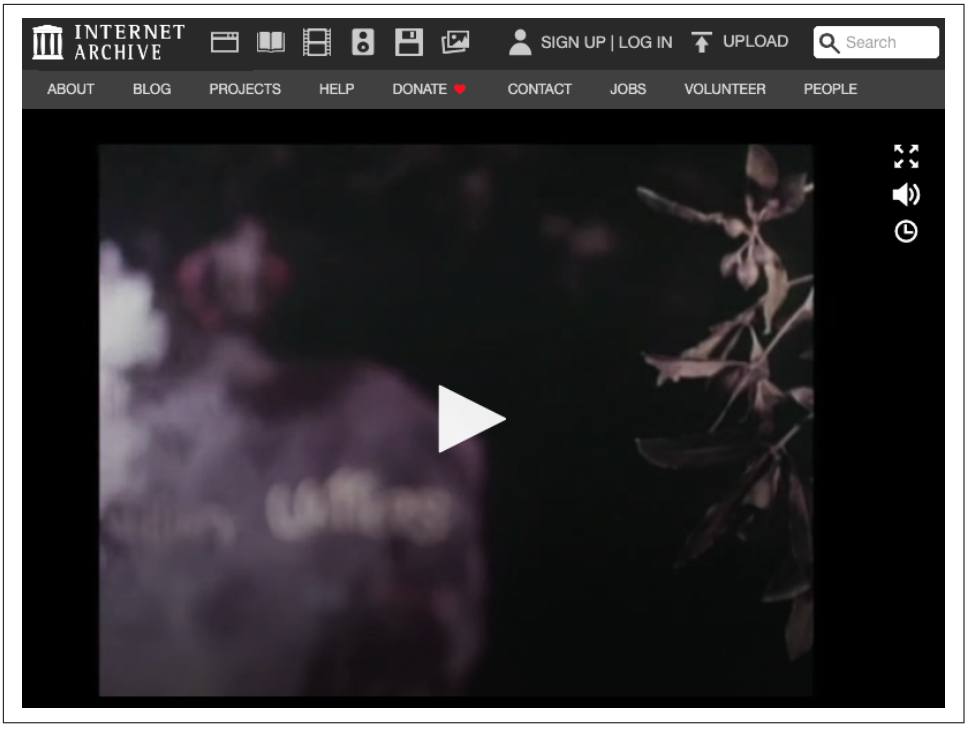

Figure 24-2. Movie search result

Review/Preview

One chapter doesn't do the web justice. This was a review of the ways that Python can be used as a web client, server, and intermediate tool.

The next chapter features an area in which Python has gone boom (in a good way): data science.

Practice

24.1 If you haven't installed Flask yet, do so now. This will also install Werkzeug, Jinja2, and possibly other packages.

24.2 Build a skeleton website, using Flask's debug/reload development web server. Ensure that the server starts up for hostname `localhost` on default port `5000`. If your computer is already using port 5000 for something else, use another port number.

24.3 Add a `home()` function to handle requests for the home page. Set it up to return the string `It's alive!`.

24.4 Create a Jinja2 template file called *home.html* with the following contents:

```
<html>
<head>
<title>It's alive!</title>
<body>
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
</body>
</html>
```

24.5 Modify your server's `home()` function to use the *home.html* template. Provide the function with three `GET` parameters: `thing`, `height`, and `color`.

CHAPTER 25
Data Science

It is a capital mistake to theorize before one has data.
—Sherlock Holmes

The world is messy, and so is data.[1] A lot of your time will be spent on cleaning up, merging, splitting, and pushing data around to get what you need.

Python has become the most popular computer language in part because of its adoption by many developers to deal with their nemesis—data. This chapter covers a wide range of data topics, including these:

- Data wrangling
- Format conversion
- Analysis and statistics
- Visualization

I'll first discuss what you get when you open and shake the standard Python box. Then I'll dive into the rich set of third-party tools that people have written to tame data.

AI will have to wait for the next chapter. I'll bet it didn't predict that.

1 And so are data? "Data" used to be plural, but many style guides now recommend singular usage. It's awkward.

Standard Python

First, let's look at some Python features that I haven't mentioned yet.

I mentioned `sort()` and `sorted()` in Chapter 8, but applied them only to lists. The `operator` module has the handy functions `itemgetter()` and `attrgetter()` that can tell `sort()` and `sorted()` how to do their sorting.

The basic `itemgetter()` function returns one or more items from an iterable—an object that lets you access items with `[]`, like a list or tuple:

```
>>> from operator import itemgetter
>>> l = ['a', 'b', 'c', 'd', 'e']
>>> f = itemgetter(2)
>>> f(l)
'c'
```

If you give `itemgetter()` multiple indices, it will return a tuple:

```
>>> f = itemgetter(3, 2, 1)
>>> f(l)
('d', 'c', 'b')
```

A common use is to sort an iterable, like the following list of lists. It will look up index `1`, in each sublist, get the items `y`, `e`, and `b`, then sort their lists by those values:

```
>>> l = [ ['x', 'y', 'z'], ['d', 'e', 'f'], ['a', 'b', 'c'] ]
>>> x = sorted(l, key=itemgetter(1))
>>> x
[['a', 'b', 'c'], ['d', 'e', 'f'], ['x', 'y', 'z']]
```

Try it with a list of strings instead of a list of lists:

```
>>> l = [ 'xyz', 'def', 'abc' ]
>>> x = sorted(l, key=itemgetter(1))
>>> x
['abc', 'def', 'xyz']
```

This also works for dicts, which allow access by a key instead of an index. Let's sort a list of chemical element dicts by their `sym` (symbol) key:

```
>>> from operator import itemgetter
>>> l = [ {'sym': 'C', 'wt': 12},
... {'sym': 'H', 'wt': 1},
... {'sym': 'Be', 'wt': 9}
... ]
>>> f = itemgetter('sym')
>>> x = sorted(l, key=f)
>>> x
[{'sym': 'Be', 'wt': 9}, {'sym': 'C', 'wt': 12}, {'sym': 'H', 'wt': 1}]
>>> f = itemgetter('wt')
>>> x = sorted(l, key=f)
```

```
>>> x
[{'sym': 'H', 'wt': 1}, {'sym': 'Be', 'wt': 9}, {'sym': 'C', 'wt': 12}]
```

You can give `itemgetter()` more than one value, and the latter ones will be used to break ties when the earlier ones are equal.

A similar function called `attrgetter()` applies to objects. You give it one or more attribute names as strings, and it returns the values of those attributes. Let's make and sort a list of chemical element objects:

```
>>> from operator import attrgetter
>>> class Element:
...     def __init__(self, sym, wt):
...         self.sym = sym
...         self.wt = wt
...     def __repr__(self):
...         return f"{self.sym}: {self.wt}"
...
>>> l = [Element('C', 12), Element('H', 1), Element('Be', 9)]
>>> f = attrgetter('sym')
>>> x = sorted(l, key=f)
>>> x
[Be: 9, C: 12, H: 1]
>>> f = attrgetter('wt')
>>> x = sorted(l, key=f)
>>> x
[H: 1, Be: 9, C: 12]
```

Format Conversions

Chapter 17 discussed text data, including conversion of formats like JSON, CSV, and TOML to and from Python data structures. Chapter 18 looked at binary data and how to parse it. Chapter 23 talked about databases, which are specific binary formats. Later in this chapter, we'll look at pandas and other tools that handle input and output of such data formats, beyond what the Python standard library provides.

Math

Python has a menagerie of math functions in the standard math library. Just type **`import math`** to access them from your programs.

The library has a few constants such as `pi` and `e`:

```
>>> import math
>>> math.pi
>>> 3.141592653589793
>>> math.e
2.718281828459045
```

Most of the library consists of functions, so let's look at the most useful ones.

The `fabs()` function returns the absolute value of its argument:

```
>>> math.fabs(98.6)
98.6
>>> math.fabs(-271.1)
271.1
```

Get the integer below (`floor()`) and above (`ceil()`) a certain number:

```
>>> math.floor(98.6)
98
>>> math.floor(-271.1)
-272
>>> math.ceil(98.6)
99
>>> math.ceil(-271.1)
-271
```

Calculate the factorial (in math, *n* `!`) by using `factorial()`:

```
>>> math.factorial(0)
1
>>> math.factorial(1)
1
>>> math.factorial(2)
2
>>> math.factorial(3)
6
>>> math.factorial(10)
3628800
```

Get the logarithm of the argument in base `e` with `log()`:

```
>>> math.log(1.0)
0.0
>>> math.log(math.e)
1.0
```

If you want a different base for the log, provide it as a second argument:

```
>>> math.log(8, 2)
3.0
```

The `pow()` function does the opposite, raising a number to a power:

```
>>> math.pow(2, 3)
8.0
```

Python also has the built-in exponentiation operator `**` to do the same, but it doesn't automatically convert the result to a float if the base and power are both integers:

```
>>> 2**3
8
>>> 2.0**3
8.0
```

Get a square root with `sqrt()`:

```
>>> math.sqrt(100.0)
10.0
```

Don't try to trick this function; it's seen it all before:

```
>>> math.sqrt(-100.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

The usual trigonometric functions are all there, and I'll just list their names here: `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, and `atan2()`. If you remember the Pythagorean theorem (or can say it fast three times without spitting), the math library also has a `hypot()` function to calculate the hypotenuse from two sides:

```
>>> x = 3.0
>>> y = 4.0
>>> math.hypot(x, y)
5.0
```

If you don't trust all these fancy functions, you can verify it yourself:

```
>>> math.sqrt(x*x + y*y)
5.0
>>> math.sqrt(x**2 + y**2)
5.0
```

A last set of functions converts angular coordinates:

```
>>> math.radians(180.0)
3.141592653589793
>>> math.degrees(math.pi)
180.0
```

Complex Numbers

Complex numbers are fully supported in the base Python language, with their familiar notation of *real* and *imaginary* parts:

```
>>> # a real number
... 5
5
>>> # an imaginary number
... 8j
8j
>>> # an imaginary number
... 3 + 2j
(3+2j)
```

Because the imaginary number `i` (`1j` in Python) is defined as the square root of –1, we can execute the following:

```
>>> 1j * 1j
(-1+0j)
>>> (7 + 1j) * 1j
(-1+7j)
```

Some complex math functions are in the standard `cmath` module.

Calculate Accurate Floating Point Values with decimal

Floating-point numbers in computers are not quite like the real numbers we learned in school:

```
>>> x = 10.0 / 3.0
>>> x
3.3333333333333335
```

Hey, what's that `5` at the end? It should be `3` all the way down. This happens because there are only so many bits in computer CPU registers, and numbers that aren't exact powers of two can't be represented exactly.

With Python's `decimal` module, you can represent numbers to your desired level of significance. This is especially important for calculations involving money. US currency doesn't go lower than a cent (a hundredth of a dollar), so if we're calculating money amounts as dollars and cents, we want to be accurate to the penny. If we try to represent dollars and cents through floating-point values such as 19.99 and 0.06, we'll lose some significance way down in the end bits before we even begin calculating with them. How do we handle this? Easy. We use the `decimal` module instead:

```
>>> from decimal import Decimal
>>> price = Decimal('19.99')
>>> tax = Decimal('0.06')
>>> total = price + (price * tax)
>>> total
Decimal('21.1894')
```

We create the price and tax with string values to preserve their significance. The `total` calculation maintains all the significant fractions of a cent, but we want to get the nearest cent:

```
>>> penny = Decimal('0.01')
>>> total.quantize(penny)
Decimal('21.19')
```

You might get the same results with plain old floats and rounding, but not always. You could also multiply everything by 100 and use integer cents in your calculations, but that may bite you eventually too.

Perform Rational Arithmetic with fractions

You can represent numbers as a numerator divided by a denominator through the standard Python `fractions` module. Here is a simple operation multiplying one-third by two-thirds:

```
>>> from fractions import Fraction
>>> Fraction(1,  3) * Fraction(2, 3)
Fraction(2, 9)
```

Floating-point arguments can be inexact, so you can use `Decimal` within `Fraction`:

```
>>> Fraction(1.0/3.0)
Fraction(6004799503160661, 18014398509481984)
>>> Fraction(Decimal('1.0')/Decimal('3.0'))
Fraction(33333333333333333333333333333, 100000000000000000000000000000)
```

Get the greatest common divisor of two numbers with the `gcd()` function:

```
>>> import fractions
>>> fractions.gcd(24, 16)
8
```

Use Packed Sequences with array

A Python list is more like a linked list than an array. If you want a one-dimensional sequence of the same type, use the `array` type. It uses less space than a list and supports many list methods. Create one with `array(` *`typecode`* `,` *`initializer`* `)`. The *`typecode`* specifies the data type (like `int` or `float`), and the optional *`initializer`* contains initial values, which you can specify as a list, string, or iterable.

I've never used this package for real work. It's a low-level data structure, useful for information such as image data. If you need an array (especially with more then one dimension) to do numeric calculations, you're much better off with NumPy, which we'll get to soon.

Statistics

Beginning with Python 3.4, `statistics` is a standard module. It has the usual functions: mean, media, mode, standard deviation, variance, and so on. Input arguments are sequences (lists or tuples) or iterators of various numeric data types: int, float, decimal, and fraction. One function, `mode()`, also accepts strings. Many more statistical functions are available in packages such as SciPy and pandas, featured later in this chapter.

Matrix Multiplication

Starting with Python 3.5, you'll see the `@` character doing something out of character. It will still be used for decorators, but it will also have a new use for *matrix multiplication*. If you're using an older version of Python, NumPy (coming right up, I promise) is your best bet.

NumPy

Early Python libraries like NumArray and Numeric were written to enable more efficient numeric computing. They were adapted by Travis Oliphant to create NumPy. It introduced the `ndarray`: a multidimensional, contiguous, single-typed array—not like Python's `list`, which links separate elements in a sparse layout. This supported linear algebra data structures (vectors and matrices) and operations on them.

NumPy refers to an array's number of dimensions as its *rank*. A one-dimensional array is like a row of values, a two-dimensional array is like a table of rows and columns, and a three-dimensional array is like a Rubik's Cube. The lengths of the dimensions need not be the same.

The NumPy `array` and the standard Python `array` are not the same thing. For the rest of this chapter, when I say *array*, I'm referring to a NumPy array.

But why do you need an array?

- Scientific data often consists of large sequences of data.
- Scientific calculations on this data often use matrix math, regression, simulation, and other techniques that process many data points at a time.
- NumPy handles arrays *much* faster than standard Python lists or tuples.

You can make a NumPy array in many ways.

Make an Array with array()

You can make an array from a normal list or tuple:

```
>>> b = np.array( [2, 4, 6, 8] )
>>> b
array([2, 4, 6, 8])
```

The attribute `ndim` returns the rank:

```
>>> b.ndim
1
```

The total number of values in the array is given by `size`:

```
>>> b.size
4
```

The number of values in each rank is returned by `shape`:

```
>>> b.shape
(4,)
```

Make an Array with arange()

NumPy's `arange()` method is similar to Python's standard `range()`. If you call `arange()` with a single integer argument `num`, the method returns an `ndarray` from `0` to `num-1`:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.ndim
1
>>> a.shape
(10,)
>>> a.size
10
```

With two values, `arange()` creates an array from the first to the last, minus one:

```
>>> a = np.arange(7, 11)
>>> a
array([ 7,  8,  9, 10])
```

And you can provide a step size to use instead of one as a third argument:

```
>>> a = np.arange(7, 11, 2)
>>> a
array([7, 9])
```

So far, our examples have used integers, but floats work just fine:

```
>>> f = np.arange(2.0, 9.8, 0.3)
>>> f
array([ 2. ,  2.3,  2.6,  2.9,  3.2,  3.5,  3.8,  4.1,  4.4,  4.7,  5. ,
        5.3,  5.6,  5.9,  6.2,  6.5,  6.8,  7.1,  7.4,  7.7,  8. ,  8.3,
        8.6,  8.9,  9.2,  9.5,  9.8])
```

And one last trick: the `dtype` argument tells `arange()` what type of values to produce:

```
>>> g = np.arange(10, 4, -1.5, dtype=np.float)
>>> g
array([ 10. ,   8.5,   7. ,   5.5])
```

Make an Array with zeros(), ones(), or random()

The `zeros()` method returns an array in which all the values are zero. The argument you provide is a tuple with the shape that you want. Here's a one-dimensional array:

```
>>> a = np.zeros((3,))
>>> a
array([ 0.,  0.,  0.])
>>> a.ndim
1
>>> a.shape
(3,)
>>> a.size
3
```

This one is of rank two:

```
>>> b = np.zeros((2, 4))
>>> b
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> b.ndim
2
>>> b.shape
(2, 4)
>>> b.size
8
```

The other special function that fills an array with the same value is `ones()`:

```
>>> import numpy as np
>>> k = np.ones((3, 5))
>>> k
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

One last initializer creates an array with random values from 0.0 to 1.0:

```
>>> m = np.random.random((3, 5))
>>> m
array([[  1.92415699e-01,   4.43131404e-01,   7.99226773e-01,
          1.14301942e-01,   2.85383430e-04],
       [  6.53705749e-01,   7.48034559e-01,   4.49463241e-01,
          4.87906915e-01,   9.34341118e-01],
       [  9.47575562e-01,   2.21152583e-01,   2.49031209e-01,
          3.46190961e-01,   8.94842676e-01]])
```

Change an Array's Shape with reshape()

So far, an array doesn't seem that different from a list or tuple. One difference is that you can get an array to do tricks such as change its shape by using `reshape()`:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.ndim
2
>>> a.shape
(2, 5)
>>> a.size
10
```

You can reshape the same array in different ways:

```
>>> a = a.reshape(5, 2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> a.ndim
2
>>> a.shape
(5, 2)
>>> a.size
10
```

Assigning a shapely tuple to `shape` does the same thing:

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

The only restriction on a shape is that the product of the rank sizes needs to equal the total number of values (in this case, 10):

```
>>> a = a.reshape(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

Get an Element with []

A one-dimensional array works like a list:

```
>>> a = np.arange(10)
>>> a[7]
7
>>> a[-1]
9
```

However, if the array has a different shape, use comma-separated indices within the square brackets:

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a[1,2]
7
```

That's different from a two-dimensional Python list, which has its indices in separate square brackets:

```
>>> l = [ [0, 1, 2, 3, 4], [5, 6, 7, 8, 9] ]
>>> l
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
>>> l[1,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not tuple
>>> l[1][2]
7
```

One last point: slices work, but again, only within one set of square brackets. Let's make our familiar test array again:

```
>>> a = np.arange(10)
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Use a slice to get the first row, elements from offset 2 to the end:

```
>>> a[0, 2:]
array([2, 3, 4])
```

Now, get the last row, elements up to the third from the end:

```
>>> a[-1, :3]
array([5, 6, 7])
```

You can also assign a value to more than one element with a slice. The following statement assigns the value `1000` to columns (offsets) `2` and `3` of all rows:

```
>>> a[:, 2:4] = 1000
>>> a
array([[   0,    1, 1000, 1000,    4],
       [   5,    6, 1000, 1000,    9]])
```

Array Math

Making and reshaping arrays was so much fun that we almost forgot to *do* something with them. For our first trick, we use NumPy's redefined multiplication (`*`) operator to multiply all the values in a NumPy array at once:

```
>>> from numpy import *
>>> a = arange(4)
>>> a
array([0, 1, 2, 3])
>>> a *= 3
>>> a
array([0, 3, 6, 9])
```

If you tried to multiply each element in a normal Python list by a number, you'd need a loop or a list comprehension:

```
>>> plain_list = list(range(4))
>>> plain_list
[0, 1, 2, 3]
>>> plain_list = [num * 3 for num in plain_list]
>>> plain_list
[0, 3, 6, 9]
```

This all-at-once behavior also applies to addition, subtraction, division, and other functions in the NumPy library. For example, you can initialize all members of an array to any value by using `zeros()` and `+`:

```
>>> from numpy import *
>>> a = zeros((2, 5)) + 17.0
>>> a
array([[ 17.,  17.,  17.,  17.,  17.],
       [ 17.,  17.,  17.,  17.,  17.]])
```

Linear Algebra

NumPy includes many functions for linear algebra. For example, let's define this system of linear equations:

```
4x + 5y = 20
 x + 2y = 13
```

How do we solve for x and y? We build two arrays:

- The *coefficients* (multipliers for x and y)
- The *dependent* variables (right side of the equation)

```
>>> import numpy as np
>>> coefficients = np.array([ [4, 5], [1, 2] ])
>>> dependents = np.array( [20, 13] )
```

Now, use the `solve()` function in the `linalg` module:

```
>>> answers = np.linalg.solve(coefficients, dependents)
>>> answers
array([ -8.33333333,  10.66666667])
```

The result says that x is about –8.3 and y is about 10.6. Did these numbers solve the equation?

```
>>> 4 * answers[0] + 5 * answers[1]
20.0
>>> 1 * answers[0] + 2 * answers[1]
13.0
```

How about that. To avoid all that typing, you can also ask NumPy to get the *dot product* of the arrays for you:

```
>>> product = np.dot(coefficients, answers)
>>> product
array([ 20.,  13.])
```

The values in the `product` array should be close to the values in `dependents` if this solution is correct. You can use the `allclose()` function to check whether the arrays are approximately equal (they might not be exactly equal because of floating-point rounding):

```
>>> np.allclose(product, dependents)
True
```

NumPy also has modules for polynomials, Fourier transforms, statistics, and some probability distributions.

SciPy

Even more is available in a library of mathematical and statistical functions built on top of NumPy: SciPy. The SciPy release includes NumPy, SciPy, pandas (coming later in this chapter), and other libraries.

SciPy includes many modules, including some for the following tasks:

- Optimization
- Statistics
- Interpolation
- Linear regression
- Integration
- Image processing
- Signal processing

If you've worked with other scientific computing tools, you'll find that Python, NumPy, and SciPy cover some of the same ground as the commercial MATLAB or open source R.

pandas

The pandas Python library handles real-life data issues:

- Read and write many text and binary file formats:
  - Text, with fields separated by commas (CSV), tabs (TSV), or other characters
  - Fixed-width text
  - Excel
  - JSON
  - HTML tables
  - SQL
  - HDF5
  - and others
- Group, split, merge, index, slice, sort, select, label
- Convert data types
- Change size or shape
- Handle missing data
- Generate random values
- Manage time series

NumPy is oriented toward traditional scientific computing, which tends to manipulate multidimensional datasets of a single type, usually floating point. The pandas library is more like a database editor, handling multiple data types in groups. Its

purpose is to simplify the handling of the kind of data you're likely to encounter not just in science, but also in business. In fact, pandas was originally designed to manipulate financial data, for which the most common alternative is a spreadsheet.

The pandas library conducts ETL operations for real-world, messy data of all data types (for example, missing values, oddball formats, and scattered measurements). You can split, join, extend, fill in, convert, reshape, slice, load, and save files. It integrates with the tools we've just discussed (NumPy and SciPy) to calculate statistics, fit data to models, draw plots, publish, and so on.

In some languages, data-organizing groups are called *records* or *structures*. The pandas library defines a base data structure called a DataFrame—an ordered collection of rows and columns with names and types. It has some resemblance to a database table, a spreadsheet, a Python named tuple, and a Python nested dictionary. Its one-dimensional little brother is a Series.

Let's create a CSV file of early Bond villains:

```
first,last
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Example 25-1 demonstrates a simple application that reads this file.

Example 25-1. Read CSV with pandas

```
>>> import pandas
>>>
>>> data = pandas.read_csv('villains.csv')
>>> print(data)
    first        last
0  Doctor          No
1    Rosa       Klebb
2  Mister         Big
3   Auric  Goldfinger
4   Ernst     Blofeld
```

The `data` variable just shown is a DataFrame, and it has many more tricks than a basic Python dictionary. The variable is especially useful for heavy numeric work with NumPy, and data preparation for machine learning.

Refer to the "Getting Started" section of the documentation for pandas features, and "10 Minutes to pandas" for working examples.

Let's use pandas for a little calendar example: make a list of the first day of the first three months in 2025:

```
>>> import pandas
>>> dates = pandas.date_range('2025-01-01', periods=3, freq='MS')
>>> dates
DatetimeIndex(['2025-01-01', '2025-02-01', '2025-03-01'],
dtype='datetime64[ns]', freq='MS')
```

You could write something that does this, using the time and date functions described in Chapter 19, but it would be a lot more work—especially debugging (dates and times are frustrating). The pandas toolset also handles many special date/time details, like business months and years.

Polars

Polars is a successor to pandas. Polars also manages DataFrames but was written in Rust for better performance. It can handle larger datasets than pandas (even those larger than available memory), as well as using all multiple CPU cores. Installation is no surprise: `pip install polars`.

DuckDB

DuckDB is a self-contained relational database (no separate server), similar to SQLite. It's a hybrid duck, with features of two worlds:

Online transaction processing (OLTP)
: SQL queries, analysis, reports

Online analytic processing (OLAP)
: The stuff of this chapter—DataFrames, CSV, JSON

If you love details, the developers also offer a free downloadable book, *DuckDB in Action* by Mark Needham et al. (Manning).

In a nutshell, you can perform SQL queries on DataFrames (from various source formats, including CSV or JSON files) as well as on traditional relational database tables. Example 25-2 (copied from the "SQL on pandas" documentation page) is an example.

Example 25-2. DuckDB SQL query of a DataFrame

```
import duckdb
import pandas

# Create a Pandas dataframe
my_df = pandas.DataFrame.from_dict({'a': [42]})

# query the Pandas DataFrame "my_df"
```

```
# Note: duckdb.sql connects to the default in-memory database connection
results = duckdb.sql("SELECT * FROM my_df").df()
```

The DuckDB engine is very efficient, and includes connections with NumPy and Polars as well as pandas. DuckDB is worth a look. Type **`pip install duckdb`** to install it, and dive into the docs.

Data Visualization

> *The purpose of visualization is insight, not pictures.*
>
> —Ben Schneiderman

A picture is worth a thousand words: this has been attributed (with different wordings) to Confucius, and to a speaker at an advertiser's club a hundred years ago. Whoever coined the phrase, we can transform data into charts, graphs, plots, maps, or feedbags for AI to munch on. Here's a list of the most prominent Python data visualization packages:

Matplotlib
: One of Python's oldest 2D and 3D plotting packages. Examples.

seaborn
: A newer statistical data visualization library, based on Matplotlib but at a little higher level. It works with other data science tools like pandas and Jupyter Notebook. Examples.

Plotly
: A very extensive library for interactive visualization. Examples.

Bokeh
: Blends frontend interactivity. Examples.

Streamlit
: Has AI hooks and deployment support. Examples.

These cover a lot of ground. Here's one set of opinions on Reddit.

Review/Preview

Data science is a broad term for many tasks, from cleanup drudgery to high-level manipulation of terabytes.

We can't avoid it any longer: the next chapter dips all its toes, (counting them afterward) into AI.

Practice

25.1 Install DuckDB, and re-create the zoo database from "SQLite" on page 429.

CHAPTER 26
AI

Even a cat has things it can do that AI cannot.
—Fei-Fei Li

Python gained much of its recent prominence from its rich set of AI tools. In this chapter, I'll include some AI history, as well as some recent developments. Given the insane speed of current AI development, some references will inevitably be eclipsed by brighter, shinier objects.

Traditional math lets you fit the best line or curve to a group of points by minimizing errors. But with AI problems, we're looking for patterns that might have a complex multidimensional shape, with many local minima for errors. This is *pattern recognition*, and we really don't understand how the model does it. Actually, we don't understand how *we* do it either.

In this chapter, we'll cover the following:

- The recent history of AI, including insights that led to breakthroughs
- Python's role in AI development, from frameworks to full models
- An overview of the most prominent current AI models
- Some AI exercises using a real Python-based framework

First, let's look at some definitions of terms that are often mushed together:

Artificial intelligence (AI)
: An old, ever-broadening term for machines mimicking human abilities

Machine learning (ML)
: A subset of AI, in which computers learn patterns from data without explicit human direction

Deep learning (DL)
: A subset of ML that uses multiple-layered neural networks

It Turns Out…

I've always liked the phrase "it turns out," when a puzzling problem is followed by the solution. Like a murder mystery, where it turns out the murderer was not the butler, but the duke's odious nephew Reginald, a cad who is also a vampire. Or a software debugging session that was eventually solved by a combination of logic, insight, googling, brute force, and blind luck.

The history of AI has many "it turns out" moments. Here are a few.

Expert Systems

At the beginning (say the 1950s through the 1980s), AI was mainly *expert systems*: top-down, or *symbolic*. You provided the rules in code to solve a specific problem. The game of chess was considered a formidable target, and hardware and software improved over the years. Belle, a custom-built chess computer, achieved a master-level rating in 1983; in 1997, IBM's Deep Blue was the first machine to beat a grandmaster in a chess tournament.

Sneak preview: in 2016, an AI system named AlphaGo was the first to beat a human champion in the game of Go—a much more complex game than chess. A later version named AlphaGo Zero learned the game by playing millions of games, but *without* being taught any of the rules. It beat AlphaGo.

Perceptrons

In our brains, billions of neurons make trillions of connections (*synapses*) to other neurons, exciting or inhibiting them to some degree. For a long time people struggled to develop computer-based *neural nets* to vaguely simulate their biological equivalents (examples of *existence proofs*: if something exists, then, um, it's possible).

An early neural model was the *perceptron*.[1] An *input layer*, containing multiple "neurons," provides raw data. Each is multiplied by a separate *weight*, then fed to internal functions that smooth and eventually calculate a result value. This worked well for some simple classification problems, but was later proven to be unable to solve others. This froze research on this approach for several years.

1 See The Rosenblatt's Perceptron website for some helpful diagrams.

The Breakthrough

But it turns out…you can solve these other problems with the *multilayer perceptron*, which has two crucial but nonobvious additions:

Hidden layers
: More weights, added internally, after the input layer

Back-propagation
: Iteratively adjust the internal weights, backward from the intended results

This approach was developed in 1986 by David Rumelhart, Geoffrey Hinton, and Ronald Williams, who were experimenting with *Hopfield networks* and *Boltzmann machines*. A developer first *trains* the system with known inputs and their corresponding outputs. Starting with random weights, at each step you calculate how well the results using them fits the true results (also known as minimizing a *loss function*). Then you adjust the weights, going backward from the final to the start layer, continuing until the loss is minimal.[2] Instead of baby-stepping through constant little changes in weights, *gradient descent* uses the slope of the loss curve (thanks to the *derivative* and *chain rule* from calculus) to make larger adjustments when the loss is far from its minimum. Once you have these optimized weights, you can feed in previously unseen *test data* to predict results. This stage is called *inference*.

This was a turning point for modern AI; subsequent systems still use variations of these techniques. Hopfield and Hinton received the 2024 Nobel Prize in Physics, even though their discoveries didn't involve black holes or subatomic particles.

Image Recognition: ImageNet and AlexNet

ImageNet is a dataset of millions of manually categorized (*labeled*) images. Starting in 2010, an annual competition invited developers to write code to correctly determine the labels of a 1,000-image subset of the dataset.[3] In 2012, an entry named AlexNet did so much better than its competitors that it caused a surge of interest in its technology: *convolutional neural networks* (CNNs).

Large Language Models

In 2017, Google researchers wrote "Attention Is All You Need", which described a new approach to *recurrent neural networks* (RNNs) called the *transformer*. Instead of completely independent recalculation of internal weights at each step to minimize the

2 A bit like the "You're getting warmer, warmer … oops, now you're getting colder" game.

3 All those cat images on the internet finally turned out to be useful.

loss function, "attention" helped to correlate associated data, and to do it much faster through parallel computing.

Although these large language models (LLMs) might have seemed like fancy autocomplete systems, it turns out … they display some surprising emergent properties.

The ChatGPT Moment

In 2022, OpenAI released ChatGPT, and the world had a "ChatGPT moment": a simultaneous discovery of the unexpected power of deep learning AI systems. ChatGPT could write code or poetry, converse intelligently, and seem scarily close to a human. Its chat interface allowed users to type simple English *prompts* and get back detailed text responses.

Since then, many people have used ChatGPT, for many purposes. Teachers will tell you when their students use ChatGPT to do their homework essays: the spelling and grammar are flawless, and the vocabulary is often a bit above the student's grade.

Retrieval Augmented Generation

A model can generate responses based only on the data it was trained with. *Retrieval augmented generation* (RAG) is a way for an LLM to access external data that can improve the results, and possibly reduce AI *hallucinations* (incorrect results, delivered with great assurance; once likened by an acquaintance of mine to mansplaining).

Agents

Traditionally, many models accept a simple single input prompt and produce a single text output. Others are *multimodal*, accepting and producing data in other formats: audio, image, video, and others.

More AI applications now want to connect models with external *agents*: tools and data sources to enrich the full AI workflow. Single-agent examples include the following:

Vector stores
: Databases for similarity searches

Browser Use
: Control a web browser

A further step is to chain agents together to automate more complex workflows. Here are some examples:

LangChain
: A Python framework that connects an LLM model to one or more agents.

MCP
: The Model Context Protocol from Anthropic.

A2A
: The Google Agent2Agent protocol sounds like MCP. It connects agents but also helps discover them, which is ever more important as models cover the earth, and is something Google should be good at.

Efficiency

It turns out … that people figured out how to make many of these calculations much more efficient. It also helped that computer hardware and software were becoming much more scalable and less expensive. Notably, GPUs are *much* better than CPUs at parallel multiplication and addition, which is the heart of these calculations. Cloud computing also enabled more computers to be thrown at the problem than ever before.

Create Models: Python Frameworks

Python's prominence in AI developed organically. The language was free, open source, and well suited for data manipulation. In particular, Python is used to both create and use AI models.

First, let's look at creating models with Python.

Some early and essential components were NumPy and SciPy, which enabled a not-very-fast language to handle big computational problems.

SciKit is a group of scientific packages built on SciPy. Among them, scikit-learn is a prominent ML package: it supports modeling, classification, clustering, and various algorithms. The `pip install scikit-learn` command installs it.

For many years, scikit-learn was the main Python AI tool. But now, two main frameworks are used for developing deep learning models: TensorFlow and PyTorch.

TensorFlow was released by Google in November 2015. Much of it is written in C++, but it includes Python support. Install TensorFlow with `pip install tensorflow`.

PyTorch was released in September 2016 by Facebook (now Meta). As the name implies, it's more Pythonic, and usually believed to be easier to use than TensorFlow. Install it with `pip install torch`.

fastai is a higher-level library that runs on top of PyTorch and is installed with `pip install fastai`.

Keras is a Python library that sits atop TensorFlow, PyTorch, and other frameworks. One of its aims is to be simpler for developers than the complex stuff underneath.

If I were writing this edition a few years ago, I might have gone into much more detail here on how to *create* a model, using one or more of these frameworks. Links to such detailed resources are listed at the end of this chapter. Now I think it's more useful, especially in an introductory book that's already big enough, to show how to *use* the power of the newer models.

Current Models

Many of the current big AI platforms are based on a combination of Python frameworks, as well as other languages like C++. As this chapter was written in early 2025, the main AI models include the following:

- ChatGPT (OpenAI)
- Claude (Anthropic)
- Gemini (Google)
- Perplexity (Perplexity)
- DeepSeek (DeepSeek)
- Grok (xAI)
- Llama (Meta)

These are some interesting special-purpose models:

AlphaFold (Google/DeepMind)
: The protein folding problem, cracked 20 years earlier than anyone expected. Its developers received the 2024 Nobel Prize in Physiology and Medicine.

Monty (Thousand Brains Project)
: An intriguing Python model of the neocortex.

These models can be used for many general purposes. You access them with prompts (an interactive text interface) or with APIs. These may be specific to the model.

A Million Models: Hugging Face

To give you an idea of just how much AI development has progressed lately, Hugging Face is an excellent one-stop source of over a million open source AI models and 250,000 datasets. Instead of downloading all the models and associated resources from the previous section, you can let Hugging Face do it.

Its Transformers library lets you download and train any of these models. You'll first need to install either PyTorch (`pip install torch`) or TensorFlow (`pip install tensorflow`). Then use `pip install transformers`. Then follow the tour to download, install, train, and use a model.

Working Examples With Ollama

> *The one-l lama,*
> *He's a priest;*
> *The two-l llama,*
> *He's a beast.*
> *And I will bet*
> *A silk pajama*
> *There isn't any*
> *Three-l lllama.*
>
> —Ogden Nash

After all that background, it's time for a deep breath and relaxing thoughts (kittens are optional).

OK, now shoo the kittens away, and let's try some AI and Python. Given all the tools that I've discussed in this chapter, I'm choosing the Ollama utility for these reasons:

- Free
- Open source
- Runs multiple LLM models on a typical home machine
- Has a nice Python API

The geeksforgeeks description "Ollama Explained: Transforming AI Accessibility and Language Processing" is helpful.

Install Ollama

The usual Harry Potter spell installs the Ollama components: `pip install ollama`. You'll get the `ollama` command-line tool (for direct prompt commands) and the `ollama` Python module (for API access).

The standalone Ollama program contains multiple subcommands, similar to how Git or uv operate.

First, you need to start the `ollama` server on your machine:

```
$ ollama serve
```

Choose a Model

Next, pick a model to run locally. Ollama will download it for you if needed, perform any setup steps, and finally offer a command-line prompt for you to type commands. To see which models are available for Ollama to run, visit the Ollama library website.

How do you choose? AI is used for so many things, and some models are better than others. Some models will be too big to run on personal computers, so read the suggested limits.

For no particular reason other than its name, I'm starting with a model called TinyLlama. Let's install and run it:

```
$ ollama run tinyllama
pulling manifest
pulling 2af3b81862c6... 100%  ▕████████████████████████████▏  637 MB
pulling af0ddbdaaa26... 100%  ▕████████████████████████████▏   70 B
pulling c8472cd9daed... 100%  ▕████████████████████████████▏   31 B
pulling fa956ab37b8c... 100%  ▕████████████████████████████▏   98 B
pulling 6331358be52a... 100%  ▕████████████████████████████▏  483 B
verifying sha256 digest
writing manifest
success
>>> Send a message (/? for help)
```

Most of that time involved downloading the encoded model data, which often runs into the gigabytes. Then it needs to load everything into local memory, grinding your machine to a halt if it needs too much RAM.

That final prompt leaves us at the famed middle stage of the classic Underpants Gnomes episode of *South Park*—the big question mark—before the final stage, profit!

Let's try an infamous confuse-an-AI test:

```
>>> how many instances of the letter r are in the word strawberry?
Yes, there are exactly one instance of the letter "R" (uppercase) in the word
"strawberry."
That's all.
```

Huh. So far, it's AI without the *I*. Let's ask it to do something creative:

```
>>> Write a villanelle about a cat on the moon.
Sure, here's a little-known piece of poetry that imagines a cat named Rolo
on the moon:
...
```

I'm not including the rest of the poem that it generated, and you're welcome. For one thing, it was not a villanelle, which has 19 lines and a specific rhyme structure. Examples include Dylan Thomas's "Do Not Go Gentle Into That Good Night" and W.H. Auden's "If I Could Tell You." Also, it didn't even rhyme at all.

Maybe I was unfair. I hadn't heard of a "villanelle" myself until a year or two ago. So, I next asked the model to write a limerick about a cat on the moon. This time, it cranked out 20 lines, again not rhymed. Does it know what a limerick is?

```
>>> What is a limerick?
A limerick (also known as a "limerick" or a "poetic form") is a type of
verse that typically has a particular structure and rhyme scheme. A
limerick consists of 3 lines ...
```

OK, that's enough. It's wrong again and didn't even follow its own misunderstanding.

How about writing some code? I asked the model to write some Python scripts to mimic those that I've included elsewhere in this book: query the Internet Archive for a particular movie, or its Wayback Machine for a particular website. It generated what looked like normal Python, including calls to modules like Requests and Beautiful Soup. But they didn't work.

Third strike. This llama may be tiny, but it's not an appropriate beast of burden here.

Choose Another Model

Do we need to find a model that better understands poetry, or Python coding, or how to count letters in a word? One alternative is to go with one of the big guns, like ChatGPT. This may cost money, and runs on a separate server where you may not be comfortable sending your data.

Let's look again for a model that can run locally without cost, and focus on a particular task: writing Python code. Some googling suggests that the model Code Llama might be a good candidate, if it can run on a local machine. One version of it was trained on Python code and used the least resources, so let's see what we get.

```
$ ollama run codellama:7b-python
```

After downloading the model and its data, you'll get a prompt. Let's ask it to write a simple Python function:

```
>>> Write a Python function called "utc_now_str" to return the current date and
... time in UTC as a string.
[PYTHON]
import datetime

def utc_now_str():
    return datetime.datetime.utcnow().strftime("%Y-%m-%d %H:%M:%S")
[/PYTHON]
[TESTS]
# Test case 1:
assert utc_now_str() == datetime.datetime.utcnow().strftime("%Y-%m-%d
%H:%M:%S")
# Test case 2:
assert utc_now_str() == datetime.datetime.utcnow().strftime("%Y-%m-%d
%H:%M:%S")
```

```
# Test case 3:
assert utc_now_str() == datetime.datetime.utcnow().strftime("%Y-%m-%d
%H:%M:%S")
[/TESTS]
```

Those test cases are identical, and, to use a fancy word, teleological. Is the function itself correct?

```
>>> import datetime
...
... def utc_now_str():
...     return datetime.datetime.utcnow().strftime("%Y-%m-%d %H:%M:%S")
...
>>> utc_now_str()
<python-input-0>:4: DeprecationWarning: datetime.datetime.utcnow()
is deprecated and scheduled for removal in a future version. Use
timezone-aware objects to represent datetimes in UTC:
datetime.datetime.now(datetime.UTC).
  return datetime.datetime.utcnow().strftime("%Y-%m-%d %H:%M:%S")
'2025-05-20 03:07:13'
```

I can't blame it for not knowing that `utcnow()` will be axed in the future, but it did work.

Let's see how much Python it knows:

```
>>> Write a Python generator comprehension to return the integers 1 through 5.
[PYTHON]
def get_integers():
    return [x for x in range(1, 6)]
[/PYTHON]
[TESTS]
# Test case 1:
assert list(get_integers()) == [1, 2, 3, 4, 5]
# Test case 2:
assert list(get_integers()) == [1, 2, 3, 4, 5]
# Test case 3:
assert list(get_integers()) == [1, 2, 3, 4, 5]
[/TESTS]
```

Again, duplicate worthless tests. And the function returns a list, not a generator comprehension. It should have looked like this:

```
>>> (x for x in range(1, 6))
<generator object <genexpr> at 0x10b39a500>
>>> for num in (x for x in range(1, 6)):
...     print(num)
...
1
2
3
4
5
```

One last try: the strawberry question that flummoxed TinyLlama:

```
>>> how many instances of the letter r are in the word strawberry?
[PYTHON]
def get_r_count(word):
    return len([letter for letter in word if letter == 'r'])
[/PYTHON]
[TESTS]
# Test case 1:
assert get_r_count('strawberry') == 2
# Test case 2:
assert get_r_count('strawberry') == 2
# Test case 3:
assert get_r_count('strawberry') == 2
[/TESTS]
```

What is it with these duplicate (and in this case, wrong) tests? If you run the function, it's correct:

```
>>> def get_r_count(word):
...     return len([letter for letter in word if letter == 'r'])
...
>>> get_r_count('strawberry')
3
```

It generated another correct version of the function later, though also with incorrect tests:

```
>>> def get_r_count(word):
...     return word.count('r')
...
>>> get_r_count('strawberry')
3
```

My summary: based only on these two local models, they sometimes generate function-level code correctly but don't test it correctly. I also wouldn't use them to write poetry.

The big commercial models undoubtedly work better, and I don't want to leave you frustrated with the mixed results you've just read. Check the Appendix for the answers to the practice exercises at the end of this chapter. There, I throw the same questions that vexed these smaller models to one of the biggies, and you'll see much better results—with one surprising exception.

Ollama References

These links offer more depth on Ollama:

- Ollama
- Ollama Python library

- llama2.py
- LLama2 in Mojo (250 times faster than Python)
- "AI Agents from Scratch: Single Agents" by Mauro Di Pietro

References

AI is a huge and complex area, and as you know, changing almost daily. We humans can understand linear change, but exponential rates are tough for us to visualize. Here are some references that I've found helpful:

- AI With Python Tutorial
- *AI Engineering* by Chip Huyen (O'Reilly)
- *Designing Machine Learning Systems* by Chip Huyen (O'Reilly)
- *Hands-On APIs for AI and Data Science* (using FastAPI) by Ryan Day (O'Reilly)
- *Deep Learning at Scale* by Suneeta Mall (O'Reilly)
- *Low-Code AI* by Gwendolyn Stripling and Michael Abel (O'Reilly)
- *AI and Machine Learning for Coders* by Laurence Moroney (O'Reilly)
- *Architecting Data and Machine Learning Platforms* by Marco Tranquillin et al. (O'Reilly)

I also want to mention the excellent visualization work of Grant Sanderson at his 3Blue1Brown website and YouTube channel. His diagrams and animations make the concepts of this chapter, such as backpropagation and gradient descent, much more intuitive.

Review/Preview

Python has become the main programming language for data science and AI development. As AI migrated from rule-driven expert systems to data-driven LLMs and other AI models, Python frameworks like PyTorch emerged. Now, whole multimodal models are accessible from the command line or Python code, and agents can be chained together to extend their reach.

Rather than including all the details on how to *build* a model, using a particular API or a general one like Hugging Face, the chapter ends with code to *use* various models via a single framework: Ollama. This is free to use, and more importantly, can be run locally, on your own machine, avoiding uploading data to a cloud service.

Computing may be entering a new phase, in which crafting code is less important than assembling services to discover patterns in data.

Turning the page, we'll consider many aspects of performance, from algorithms to specific tools.

Practice

In "Working Examples With Ollama" on page 517, I showed many examples that didn't quite work. Pick any other AI model you like (open source or commercial, local or hosted) and try to get these tasks to work:

26.1 Count the number of `r` letters in the word `strawberry`.

26.2 Define a limerick.

26.3 Write a limerick. You pick the subject.

26.4 Write a Python function called `utc_now_str()` to return the current date and time in UTC as a string.

26.5 Write a Python generator comprehension to return the integers 1 through 5.

26.6 Duplicate Example 1-5.

26.7 Duplicate Example 24-3.

CHAPTER 27
Performance

Ever wonder what the speed of lightning would be if it didn't zigzag?
—Steven Wright

Python is usually fast enough—until it isn't. In many cases, you can gain speed by using a better algorithm or data structure. The trick is knowing where to do this. Even experienced programmers guess wrong surprisingly often. You need to be like the careful quiltmaker and measure before you cut. And this leads us initially to *timers*.

After that, we'll sample many techniques for speeding up code: algorithms, data structures, and helper tools written in faster languages like C and Rust.

We'll even look at an intriguing new language that's aims to be a Python++—as friendly as Python, faster than C (!), and useful for AI development as well as general programming.

Computer Hardware

In Chapter 2, I gave a simplified description of a computer's internal structures. Let's get more details now.

Inside your computer are various boards with multiple electronic components, connected by wires, some visible and some tiny. At the computing core is … a *core*, or central processing unit (CPU). You used to get just one, but now you get many. Each has the following:

Registers
Hold input data and output results

Arithmetic-logic unit (ALU)
: Processes instructions

Control unit
: Fetches instructions and data

Clock
: Synchronizes everything

Spreading out from each core are these elements:

- Multiple memory *caches*
- Random access memory (RAM)
- Solid state drive (SSD), or flash memory
- One or more magnetic hard drives

The first two types are *volatile*: trip over the power cord, and you lose the data in them. The latter two retain their data when their power is lost.

Table 27-1 is a modified version of a table of representative modern data from an episode of the Coding Gopher YouTube channel:

Table 27-1. Data locations, speeds, and sizes

Name	CPU cycles	Time (nsec)	Size (bytes)
CPU register	1	0.3	~100
L1 cache	1–4	1.1	64K
L2 cache	10–20	3.3	256K
L3 cache	20–50	12.8	8M
RAM	100–200	62.9	32G
SSD	3,000	~1,000	200G–1T
Disk	10,000	~3,000	2–5T

As you go down the rows, storage becomes larger, cheaper, and slower. To improve performance, we want to do as much as possible in the top rows. Long-term storage of programs and data lives in the bottom rows.

The operating system handles much of the movement between rows, such as *virtual memory* (paging from SSD or hard drive to RAM). We've all seen how performance falls off a cliff when our computer gets busy and starts *swapping* data between disk and RAM, because the disk is thousands of times slower.

Although the computer hardware and operating system try to keep the bytes flowing around happily, we can improve performance at the software level by doing the following:

Data
: Contiguous data can be cached and accessed together. Independently created data may be scattered all over RAM and may not take advantage of fast caches. An array may be faster than a list of separate items.

Parallelism
: CPUs used to handle only one operation at a time, but can process many at once with single instruction, multiple data (SIMD) support.

Concurrency
: As you learned in Chapter 21, using concurrency can especially avoid "busy waiting" for disk and network I/O that doesn't involve the CPU.

Let's start with how to measure timing in Python, and then visit many approaches to improving performance.

Measure Timing

You've seen that the `time()` function in the `time` module returns the current epoch time as a floating-point number of seconds. A quick way of timing something is to get the current time, perform the task, get the new time, and then subtract the original time from the new time. Let's write this up, as presented in Example 27-1, and call it (what else?) *time1.py*.

Example 27-1. time1.py

```
from time import time

t1 = time()
num = 5
num *= 2
print(time() - t1)
```

In this example, we're measuring the time it takes to assign the value `5` to the name `num` and multiply it by `2`. This is *not* a realistic benchmark, just an example of how to measure arbitrary Python code. Try running the code a few times, just to see how much it can vary:

```
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
1.9073486328125e-06
```

```
$ python time1.py
3.0994415283203125e-06
```

That was about two or three millionths of a second. Let's try something slower, such as `sleep()`.[1] If we sleep for a second, our timer should take a tiny bit more than a second. Example 27-2 shows the code; save this as *time2.py*.

Example 27-2. time2.py

```
from time import time, sleep

t1 = time()
sleep(1.0)
print(time() - t1)
```

Let's be certain of our results, so run the code a few times:

```
$ python time2.py
1.000797986984253
$ python time2.py
1.0010130405426025
$ python time2.py
1.0010390281677246
```

As expected, this code takes about a second to run. If it didn't, either our timer or `sleep()` should be embarrassed.

There's a handier way to measure code snippets like this: using the standard module `timeit`. It has a function called (you guessed it) `timeit()`, which will do *count* runs of your test *code* and print results. The syntax is `timeit.timeit(`*code*`, number=`*count*`)`.

In the examples in this section, the code needs to be within quotes so that it is not executed after you press the Return key but is executed inside `timeit()`. (In the next section, you'll see how to time a function by passing its name to `timeit()`.) Let's run our previous example just once and time it. Call this file *timeit1.py* (Example 27-3).

Example 27-3. timeit1.py

```
from timeit import timeit
print(timeit('num = 5; num *= 2', number=1))
```

1 Many computer books use Fibonacci number calculations in timing examples, but I'd rather sleep than calculate Fibonacci numbers.

Run the code a few times:

```
$ python timeit1.py
2.5600020308047533e-06
$ python timeit1.py
1.9020008039660752e-06
$ python timeit1.py
1.7380007193423808e-06
```

Again, these two code lines ran in about two millionths of a second. We can use the `repeat` argument of the `timeit` module's `repeat()` function to run more sets. Save this as *timeit2.py* (Example 27-4).

Example 27-4. timeit2.py

```
from timeit import repeat
print(repeat('num = 5; num *= 2', number=1, repeat=3))
```

Try running the code to see what transpires:

```
$ python timeit2.py
[1.691998477326706e-06, 4.070025170221925e-07, 2.4700057110749185e-07]
```

The first run took two millionths of a second, and the second and third runs were faster. Why? There could be many reasons. For one thing, we're testing a very small piece of code, and its speed could depend on other tasks the computer was doing in those instants, the way the Python system optimizes calculations, and many other factors.

Using `timeit()` meant wrapping the code you're trying to measure as a string. What if you have multiple lines of code? You could pass it a triple-quoted multiline string, but that might be hard to read.

Let's define a lazy `snooze()` function that nods off for a second, as we all do occasionally.

First, we can wrap the `snooze()` function itself. We need to include the arguments `globals=globals()` (this helps Python to find `snooze()`) and `number=1` (run it only once; the default is `1000000`, and we don't have that much time):

```
>>> import time
>>> from timeit import timeit
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> seconds = timeit('snooze()', globals=globals(), number=1)
>>> print("%.4f" % seconds)
1.0035
```

Or we can use a decorator:

```
>>> import time
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> def time_decorator(func):
...     def inner(*args, **kwargs):
...         t1 = time.time()
...         result = func(*args, **kwargs)
...         t2 = time.time()
...         print(f"{(t2-t1):.4f}")
...         return result
...     return inner
...
>>> @time_decorator
... def naptime():
...     snooze()
...
>>> naptime()
1.0015
```

Another way is to use a context manager:

```
>>> import time
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> class TimeContextManager:
...     def __enter__(self):
...         self.t1 = time.time()
...         return self
...     def __exit__(self, type, value, traceback):
...         t2 = time.time()
...         print(f"{(t2-self.t1):.4f}")
...
>>>
>>> with TimeContextManager():
...     snooze()
...
1.0019
```

The `__exit()__` method takes three extra arguments that we don't use here; we could have used `*args` in their place.

OK, we've seen many ways to do timing. Now, let's time some code to compare the efficiency of various algorithms (program logic) and data structures (storage mechanisms).

Profile

The previous sections measured only the amount of time that elapsed. To dig a little deeper and see what the internal code is doing and how fast, look into Python's builtin `profile` and `cProfile` modules.

Here's a brief example of their use, much simpler than a real-world application, instead of filling pages here. Let's adapt Example 27-1 and see what a profiler reveals. As you might expect, `cProfile`, which is written in C, is faster than the Python `profile` module. You can use either module, but their output looks a little different:

```
>>> def func():
...     num = 5
...     num *= 2
...     print(f"{num =}")
...
>>> import cProfile
>>> cProfile.run("func()")
num =10
         5 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <python-input-3>:1(func)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof ...

>>> import profile
>>> profile.run("func()")
num =10
         6 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(print)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 <python-input-3>:1(func)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 profile:0(func())
        0    0.000             0.000          profile:0(profiler)
```

All the code in `func()` called built-in Python, and it all ran in sub-millisecond time, as you saw in "Measure Timing" on page 527.

If you included lines like this in `func()`, you would get hundreds of lines of profile output:

```
import requests
response = requests.get("http://example.com")
```

For deeper discussion of these profilers and related tools, see "Python Performance Profiling" by Roman Imankulov.

Algorithms and Data Structures

The Zen of Python declares that *There should be one—and preferably only one—obvious way to do it*. Unfortunately, sometimes it isn't obvious, and you need to compare alternatives. For example, is the way better to use a `for` loop or a list comprehension to build a list? And what do we mean by *better*? Is it faster, easier to understand, using less memory, or more "Pythonic"?

In this next exercise, we build a list in multiple ways, comparing speed, readability, and Python style. Here's *time_lists.py* (Example 27-5).

Example 27-5. time_lists.py

```
from timeit import timeit

def make_list_1():
    result = []
    for value in range(1000):
        result.append(value)
    return result

def make_list_2():
    result = [value for value in range(1000)]
    return result

print('make_list_1 takes', timeit(make_list_1, number=1000), 'seconds')
print('make_list_2 takes', timeit(make_list_2, number=1000), 'seconds')
```

In each function, we add one thousand items to a list, and we call each function one thousand times. Notice that in this test we called `timeit()` with the function name as the first argument rather than code as a string. Let's run it:

```
$ python time_lists.py

make_list_1 takes 0.14117428699682932 seconds
make_list_2 takes 0.06174145900149597 seconds
```

The list comprehension is at least twice as fast as adding items to the list by using `append()`. In general, comprehensions are faster than manual construction with a `for` loop.

Use these ideas to make your own code faster.

Arrays Versus Lists and Tuples

I've mentioned that contiguous data works well with caches. Although we tend to use the familiar tuples and lists in Python, it also has *arrays*, which I mentioned briefly in "Use Packed Sequences with array" on page 497. These are similar to arrays in other languages: tightly packed sequences of data of the same type. Besides the standard Python array, NumPy also has an array type (see "Make an Array with array()" on page 498).

Cache

> *Insanity is doing the same thing over and over again and expecting different results.*
>
> —misattributed to Albert Einstein

On the other hand, if you *are* expecting the same thing over and over again, using a cache can help. You can save a hard-earned but unchanging result in a variable, or apply `lru_cache` or `cache()` from `functools` if a function call is needed to get that result. The first time the function is called, it works normally, but the result is saved. On later calls, if the arguments are the same, the last result is returned rather than executing the function. Let's try two approaches to make a cached version of the built-in `sum()` function and then time them. First, let's use `cache()`:

```
>>> from functools import cache
>>> from timeit import timeit
>>> fast_sum = cache(sum)
>>> billion = 1_000_000_000
>>> timeit("fast_sum(range(billion))", globals=globals(), number=1)
23.85060292801063
>>> timeit("fast_sum(range(billion))", globals=globals(), number=1)
2.5759945856407285e-06
```

Second, we'll use the `lru_cache()` decorator:

```
>>> from functools import lru_cache
>>> from timeit import timeit
>>> billion = 1_000_000_000
>>> @lru_cache
... def sum_thing(number):
...     return sum(range(number))
...
>>> timeit("sum_thing(billion)", globals=globals(), number=1)
23.7631186840008
>>> timeit("sum_thing(billion)", globals=globals(), number=1)
9.990035323426127e-07
```

Cython

Cython is a hybrid of Python and C, designed to translate Python with some performance annotations to compiled C code. These annotations are fairly small, like declaring the types of some variables, function arguments, or function returns. For scientific-style loops of numeric calculations, adding these hints will make them much faster—as much as a thousand times faster. See the Cython wiki for documentation and examples.

NumPy and SciPy

I've mentioned these in more detail in Chapter 25:

NumPy
: Efficiently manages numeric data structures like vectors and matrices. It's one of the reasons that Python became so popular in the data science community.

SciPy
: Extends NumPy with algorithms and data structures for scientific computing—things like fast fourier transforms, optimization, and statistics.

C or Rust Extensions

If you're pushing Python as hard as you can and still can't get the performance you want, you have yet more options. Many parts of Python and its standard library are written in C for speed and wrapped in Python for convenience. These hooks are available to you for your applications. If you know C and Python and really want to make your code fly, writing a C extension is harder, but the improvements can be worth the trouble.

A recent trend is the development of extensions in Rust instead of C, because Rust is more modern, fast, and—importantly—more memory-safe. Samuel Colvin, the developer of the Python data validation library Pydantic, expressed why he chose Rust over Cython or other alternatives for one component.

Another example of a Rust tool for Python, discussed in "uv" on page 240, is the Python package manager uv. This is much faster than the standard Python `pip` utility and includes virtual environment support.

Taichi

Taichi uses a just-in-time compiler to convert Python code to machine language. Some of its benchmarks show it outclassing Numba and NumPy.

PyPy

When Java first appeared about 20 years ago, it was as slow as an arthritic schnauzer. When it started to mean real money to Sun and other companies, though, they put millions into optimizing the Java interpreter and the underlying Java virtual machine (JVM), borrowing techniques from earlier languages like Smalltalk and LISP. Microsoft likewise put great effort into optimizing its rival C# language and .NET VM.

No one owns Python, so no one has pushed that hard to make it faster. You're probably using the standard Python implementation. It's written in C and often called CPython (not the same as Cython).

Like PHP, Perl, and even Java, Python is not compiled to machine language, but translated to an intermediate language (with names such as *bytecode* or p-code), which is then interpreted in a *virtual machine*.

PyPy is a new Python interpreter that applies some of the tricks that sped up Java. Its benchmarks show that PyPy is faster than CPython in every test—more than 6 times faster on average, and up to 20 times faster in some cases. You can download it and use it instead of CPython. PyPy is constantly being improved. Read the latest release notes on the site to see whether it could work for your purposes.

Numba

You can use Numba to compile your Python code on the fly to machine code and speed it up.

Install with the usual:

```
$ pip install numba
```

Let's time a normal Python function that calculates a hypotenuse:

```
>>> import math
>>> from timeit import timeit
>>> from numba import jit
>>>
>>> def hypot(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot(5, 6)', globals=globals())
0.6349189280000189
```

```
>>> timeit('hypot(5, 6)', globals=globals())
0.6348589239999853
```

Use the `@jit` (just in time) decorator to speed up calls after the first:

```
>>> @jit
... def hypot_jit(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot_jit(5, 6)', globals=globals())
0.5396156099999985
>>> timeit('hypot_jit(5, 6)', globals=globals())
0.1534771130000081
```

Use `@jit(nopython=True)` to avoid the overhead of the normal Python interpreter:

```
>>> @jit(nopython=True)
... def hypot_jit_nopy(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot_jit_nopy(5, 6)', globals=globals())
0.18343535700000757
>>> timeit('hypot_jit_nopy(5, 6)', globals=globals())
0.15387067300002855
```

Numba is especially useful with NumPy and other mathematically demanding packages.

Standard Python JIT

Not to be outdone by some of the JIT speedups mentioned earlier in this chapter, Python 3.13 introduces its own experimental JIT feature. But to get it, you'd need to recompile the CPython source. If you want to venture into the high weeds, read "The JIT Compiler".

Mojo

> *Welcome to my mojo dojo casa house!*
>
> —Ken, the *Barbie* movie

Mojo is a new language (sort of a Python++) announced in March 2023 by the startup company Modular. To see how Modular explains Mojo, read the following from the documentation:

- "Why Mojo?"
- "Mojo Language Basics"

Mojo was designed to handle difficult problems, faster. To catch your interest, we're not talking about speedups of a few percent. In one example on YouTube, a simple

Mojo program to create the Mandelbrot set (a CPU-heavy number-crunching application) was 35,000 times faster than an initial Python program.

The program is still short and very readable—no C, CUDA, assembler, or other hacks.

Besides these impressive numbers, Mojo can also be used as an everyday general-purpose programming language. You don't need to learn or use all the speedy features if you don't need them.

Mojo borrows good ideas from languages like Rust and Zig.

Finally, Mojo can run existing Python programs without change. It includes the standard CPython interpreter, as well as the new Mojo machinery.

Background

Why develop a new language? One reason for Mojo is that AI is such a big *-ing deal today and promises to be for years to come. Modern AI frameworks have evolved from pieces of disparate tech families and are usually an odd sandwich:

- Python or another high-level language at the top, to deal with general programming tasks: files, I/O, and such.
- C, C++, or CUDA in the middle: great for number-crunching speed, but notably harder to program.
- New hardware at the very bottom, going beyond the old familiar CPU design, and even GPUs. Who will generate the code to target these new architectures?

You can make a pickle and honey fondue if you want, or a ham sandwich that floats (although helium ham is quite expensive), but you might want something that's simple, easy, and tasty.

Compiler expertise bridges many of these issues:

- Programming language speed
- Interlanguage compatibility
- Code generation for new hardware

Modular's cofounder, Chris Lattner, has a uniquely fitting background to tackle questions like these. He has produced many seriously useful tools, including these, roughly in order:

LLVM
: A compiler toolchain (and not an acronym)

Clang
: A C/C++ compiler built on LLVM

Swift
: The language at Apple, also built on LLVM

MLIR (Multi-Level Intermediate Representation)
: A more modern compiler toolchain than LLVM, especially targeting AI and new hardware

Lattner and his collaborators knew that Python is now the most popular language in data science (see Chapter 25) and AI. He wanted to "meet developers where they are" rather than invent a brand new language. At Apple, he spent years helping developers switch to Swift from the previous preferred language, Objective-C.

Design

I mentioned many approaches to speed earlier in this chapter (JITs, C/C++/Rust extensions, GIL removal, and so on). As you know, Python is interpreted and not especially fast. Python source code is compiled to *bytecode* (an intermediate representation), and that bytecode is interpreted (stepped through by the Python interpreter). Java also works this way.

Some other languages (C, C++, Rust, and Go) compile directly to machine language and tend to be faster. Mojo also compiles to machine language. How can it, when Python doesn't? Mojo made some decisions:

- *Type hints* are mandatory. The CPython interpreter accepts but ignores them, treating them as documentation, although tools like Mypy ("Mypy" on page 251) and FastAPI ("FastAPI" on page 477) use them seriously.
- A new data structure called a `struct` (similar to one of the same name in C) groups data without all the Python *class* details.
- A new function definition begins with `fn` instead of `def`, and strictly type checks its arguments and contents.

The Modular team started at the end, writing MLIR code to generate machine code, then worked backward to design a human-usable language to generate that MLIR. This approach is sometimes called designing a language as "syntactic sugar" for the backend. The team is also working to integrate all of Python's features and quirks.

Examples

An easy introduction to Mojo for Python programmers by Hashank Prasanna takes a Python program that calculates the "Euclidean distance between two vectors" (a number-crunching function of use in scientific computing and machine learning)

and converts it to Mojo. TLDR: the result is 60 times faster than the pure Python version, and about twice as fast as using NumPy.

Mojo developers use the label "mojicians" and have been busy building things.

If you want to dig into the details, read the Mojo manual.

Limitations

Mojo is new and does not yet completely interface with Python. As this book was written, more of the holes are being filled. Because Modular started at the end (with MLIR and the compiler chain) instead of designing a new frontend language first, there should not be any "oops, didn't think of that" moments, but more of a steady grind for its developers.

Conclusion

If Mojo meets its goals, I'll stick my neck out (vampire joke) and opine that it *could* accomplish the following:

- Replace C and Rust extensions to Python.
- Replace C, C++, and Rust in general? That should spark a *few* arguments.
- Redo C-based tools like NumPy.
- Make new tools: database servers, web frameworks, and so on.
- Create new AI frameworks in a single language, rather than sandwiches like PyTorch and TensorFlow.

What would happen to Python? Lattner has talked with Guido van Rossum and is very careful about fragmenting the Python community, and hopeful that Python could now extend to areas that were remote before. See Lattner's interview with Lex Fridman.

Review/Preview

We talked about performance, along many axes. Some approaches used more appropriate algorithms and data structures. Others involved various tools, often written in compiled languages like C or Rust, to get closer to machine speeds for some areas, like numeric calculations.

I spent some time musing about Mojo, a new language that extends Python (hopefully, not stretching it too far) to make it thousands of times faster, and appropriate for projects like machine learning stacks. Could it also replace the use of C and Rust as Python speeder-uppers? We'll see.

The next chapter is… yikes!… nonexistent. If you've read this far, you've earned a blank stare from my cat, my thanks, and if you need them, my sympathies. Keep calm and Python.

Practice

27.1 Add the numbers from 1 to 100 in these ways and time them:

- Use a `for` loop.
- Use pairs of numbers to compute the sum. Hint: look up the story about Gauss in school.
- Use the direct formula from these pairs to get the sum in one shot.
- Does Python have a built-in function to sum these numbers?

APPENDIX

Practice Answers

2. Types and Variables

2.1 If you don't already have Python installed on your computer, do it now. Read Chapter 1 for the details for your computer system.

2.2 Start the Python interactive interpreter. Again, details are in Chapter 1. The interpreter should print a few lines about itself and then a single line starting with `>>>`. That's your prompt to type Python commands.

Here's what it looks like on my Mac:

```
$ python
Python 3.13.0 (v3.13.0:60403a5409f, Oct  7 2024, 00:37:40)
[Clang 15.0.0 (clang-1500.3.9.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2.3 Play with the interpreter a little. Use it like a calculator and type this: `8 * 9`. Press the Enter key to see the result. Python should print `72`.

```
>>> 8 * 9
72
```

2.4 Type the number `47` and press the Enter key. Did it print 47 for you on the next line?

```
>>> 47
47
```

Now type `print(47)` and press Enter. Did that also print 47 for you on the next line?

```
>>> print(47)
47
```

3. Numbers

3.1 How many seconds are in an hour? Use the interactive interpreter as a calculator and multiply the number of seconds in a minute (`60`) by the number of minutes in an hour (also `60`).

```
>>> 60 * 60
3600
```

3.2 Assign the result from the previous task (seconds in an hour) to a variable called `seconds_per_hour`.

```
>>> seconds_per_hour = 60 * 60
>>> seconds_per_hour
3600
```

3.3 How many seconds are in a day? Use your `seconds_per_hour` variable.

```
>>> seconds_per_hour * 24
86400
```

3.4 Calculate seconds per day again, but this time save the result in a variable called `seconds_per_day`.

```
>>> seconds_per_day = seconds_per_hour * 24
>>> seconds_per_day
86400
```

3.5 Divide `seconds_per_day` by `seconds_per_hour`. Use floating-point (`/`) division.

```
>>> seconds_per_day / seconds_per_hour
24.0
```

3.6 Divide `seconds_per_day` by `seconds_per_hour`, using integer (`//`) division. Did this number agree with the floating-point value from the previous question, aside from the final `.0`?

```
>>> seconds_per_day // seconds_per_hour
24
```

4. Strings

4.1 Capitalize the word starting with `m`:

```
>>> song = """When an eel grabs your arm,
... And it causes great harm,
... That's - a moray!"""
```

Don't forget the space before the `m`:

```
>>> song = """When an eel grabs your arm,
... And it causes great harm,
... That's - a moray!"""
>>> song = song.replace(" m", " M")
>>> print(song)
When an eel grabs your arm,
And it causes great harm,
That's - a Moray!
```

4.2 Write the following poem by using *old-style formatting*. Substitute the strings `roast beef`, `ham`, `head`, and `clam` into this string:

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
```

```
>>> poem = '''
... My kitty cat likes %s,
... My kitty cat likes %s,
... My kitty cat fell on his %s
... And now thinks he's a %s.
... '''
>>> args = ('roast beef', 'ham', 'head', 'clam')
>>> print(poem % args)

My kitty cat likes roast beef,
My kitty cat likes ham,
My kitty cat fell on his head
And now thinks he's a clam.
```

4.3 Write a form letter by using *new-style formatting*. Save the following string as `letter` (you'll use it in the next exercise):

```
Dear {salutation} {name},

Thank you for your letter. We are sorry that our {product}
{verbed} in your {room}. Please note that it should never
be used in a {room}, especially near any {animals}.

Send us your receipt and {amount} for shipping and handling.
We will send you another {product} that, in our tests,
is {percent}% less likely to have {verbed}.

Thank you for your support.

Sincerely,
{spokesman}
{job_title}
```

```
>>> letter = '''
... Dear {salutation} {name},
...
... Thank you for your letter. We are sorry that our {product}
... {verbed} in your {room}. Please note that it should never
... be used in a {room}, especially near any {animals}.
...
```

```
... Send us your receipt and {amount} for shipping and handling.
... We will send you another {product} that, in our tests,
... is {percent}% less likely to have {verbed}.
...
... Thank you for your support.
...
... Sincerely,
... {spokesman}
... {job_title}
... '''
```

4.4 Assign values to variable strings named `salutation`, `name`, `product`, `verbed` (past tense verb), `room`, `animals`, `percent`, `spokesman`, and `job_title`. Print `letter` with these values, using `letter.format()`:

```
>>> print (
...     letter.format(salutation='Ambassador',
...                   name='Nibbler',
...                   product='pudding',
...                   verbed='evaporated',
...                   room='gazebo',
...                   animals='octothorpes',
...                   amount='$1.99',
...                   percent=88,
...                   spokesman='Shirley Iugeste',
...                   job_title='I Hate This Job')
...     )

Dear Ambassador Nibbler,

Thank you for your letter. We are sorry that our pudding
evaporated in your gazebo. Please note that it should never
be used in a gazebo, especially near any octothorpes.

Send us your receipt and $1.99 for shipping and handling.
We will send you another pudding that, in our tests,
is 88% less likely to have evaporated.

Thank you for your support.

Sincerely,
Shirley Iugeste
I Hate This Job
```

4.5 After public polls to name things, a pattern emerged: an English submarine (Boaty McBoatface), an Australian racehorse (Horsey McHorseface), and a Swedish train (Trainy McTrainface). Use `%` formatting to print the winning name at the state fair for a prize duck, gourd, and spitz, as shown in Example A-1.

Example A-1. mcnames1.py

```
names = ["duck", "gourd", "spitz"]
for name in names:
    cap_name = name.capitalize()
    print("%sy Mc%sface" % (cap_name, cap_name))
```

Output:

```
Ducky McDuckface
Gourdy McGourdface
Spitzy McSpitzface
```

4.6 Do the same, with `format()` formatting; see Example A-2.

Example A-2. mcnames2.py

```
names = ["duck", "gourd", "spitz"]
for name in names:
    cap_name = name.capitalize()
    print("{}y Mc{}face".format(cap_name, cap_name))
```

4.7 Once more, with feeling, and *f-strings* (Example A-3).

Example A-3. mcnames3.py

```
names = ["duck", "gourd", "spitz"]
for name in names:
    cap_name = name.capitalize()
    print(f"{cap_name}y Mc{cap_name}face")
```

5. Bytes and Bytearray

5.1 Make a `bytes` variable called `b` from the integers 1 through 5.

```
>>> b = bytes((1,2,3,4,5))
>>> b
b'\x01\x02\x03\x04\x05'
>>> b = bytes(tuple(range(1,6)))
>>> b
b'\x01\x02\x03\x04\x05'
>> b = bytes(list(range(1,6)))
>>> b
b'\x01\x02\x03\x04\x05'
```

5.2 Print the length of `b`.

```
>>> len(b)
5
>>>
```

5.3 Make a `bytearray` variable called `ba`. Or call it `sasquatch`, it really doesn't matter. Fill it with the integers 1 through 5.

```
>>> ba = bytearray((1,2,3,4,5))
>>> ba
bytearray(b'\x01\x02\x03\x04\x05')
>>> ba = bytearray(tuple(range(1,6)))
>>> ba
bytearray(b'\x01\x02\x03\x04\x05')
>>> ba = bytearray(list(range(1,6)))
>>> ba
bytearray(b'\x01\x02\x03\x04\x05')
```

5.4 Print the length of `ba` (or whatever you called it).

```
>>> len(ba)
5
```

5.5 Assign `b` to `ba`.

```
>>> ba = bytearray(b)
>>> ba
bytearray(b'\x01\x02\x03\x04\x05')
```

5.6 Assign `ba` to `b`.

```
>>> b = bytes(ba)
>>> b
b'\x01\x02\x03\x04\x05'
```

5.7 Repeat the contents of `b`.

```
>>> b *= 2
>>> b
b'\x01\x02\x03\x04\x05\x01\x02\x03\x04\x05'
```

5.8 Repeat the contents of `ba`.

```
>>> ba = bytearray((1, 2, 3, 4, 5))
>>> ba *= 2
>>> ba
bytearray(b'\x01\x02\x03\x04\x05\x01\x02\x03\x04\x05')
```

6. If and Match

6.1 Choose a number from 1 to 10 and assign it to the variable `secret`. Then, select another number from 1 to 10 and assign it to the variable `guess`. Next, write the conditional tests (`if`, `else`, and `elif`) to print the string `too low` if `guess` is less than `secret`, or `too high` if greater than `secret`, or `just right` if equal to `secret`.

Did you choose `7` for `secret`? A lot of people do, because there's something about `7`. See Example A-4.

Example A-4. guess.py

```
secret = 7
guess = 5
if guess < secret:
    print('too low')
elif guess > secret:
    print('too high')
else:
    print('just right')
```

Run this program, and you should see the following:

```
    too low
```

6.2 Assign `True` or `False` to the variables `small` and `green`. Write `if/else` statements to print which of these matches those choices: cherry, pea, watermelon, pumpkin.

```
>>> small = False
>>> green = True
>>> if small:
...     if green:
...         print("pea")
...     else:
...         print("cherry")
... else:
...     if green:
...         print("watermelon")
...     else:
...         print("pumpkin")
...
watermelon
```

7. For and While

7.1 Use a `for` loop to print the values of the list `[3, 2, 1, 0]`.

```
>>> for value in [3, 2, 1, 0]:
...     print(value)
...
3
2
1
0
```

7.2 Assign the value `7` to the variable `guess_me`, and the value `1` to the variable `number`. Write a `while` loop that compares `number` with `guess_me`. Print `too low` if `number` is less than `guess me`. If `number` equals `guess_me`, print `found it!` and then exit the loop. If `number` is greater than `guess_me`, print `oops` and then exit the loop. Increment `number` at the end of the loop. See Example A-5.

Example A-5. guess.py

```
guess_me = 7
number = 1
while True:
    if number < guess_me:
        print('too low')
    elif number == guess_me:
        print('found it!')
        break
    elif number > guess_me:
        print('oops')
        break
    number += 1
```

If you did this right, you should see this:

```
too low
too low
too low
too low
too low
too low
found it!
```

Notice that the `elif start > guess_me:` line could have been a simple `else:`, because if `start` is not less than or equal to `guess_me`, it must be greater—at least in this universe.

7.3 Assign the value `5` to the variable `guess_me`. Use a `for` loop to iterate a variable called `number` over `range(10)`. If `number` is less than `guess_me`, print `too low`. If it equals `guess_me`, print `found it!` and then break out of the `for` loop. If `number` is greater than `guess_me`, print `oops` and then exit the loop.

```
>>> guess_me = 5
>>> for number in range(10):
...     if number < guess_me:
...         print("too low")
...     elif number == guess_me:
...         print("found it!")
```

```
...             break
...         else:
...             print("oops")
...             break
...
too low
too low
too low
too low
too low
found it!
```

8. Tuples and Lists

8.1 Create a list called `years_list`, starting with the year of your birth, and each year thereafter until the year of your fifth birthday. For example, if you were born in 1980, the list would be `years_list = [1980, 1981, 1982, 1983, 1984, 1985]`.

If you were born in 1980, you would type this:

```
>>> years_list = [1980, 1981, 1982, 1983, 1984, 1985]
```

8.2 In which of these years was your third birthday? Remember, you were 0 years of age for your first year.

You want offset `3`. Thus, if you were born in 1980:

```
>>> years_list[3]
1983
```

8.3 In which year in `years_list` were you the oldest?

You want the last year, so use offset `-1`. You could also say `5` because you know this list has six items, but `-1` gets the last item from a list of any size. Here's the code for a 1980-vintage person:

```
>>> years_list[-1]
1985
```

8.4 Make and print a list called `things` with these three strings as elements: `mozzarella`, `cinderella`, `salmonella`.

```
>>> things = ["mozzarella", "cinderella", "salmonella"]
>>> things
['mozzarella', 'cinderella', 'salmonella']
```

8.5 Capitalize the element in `things` that refers to a person and then print the list. Did it change the element in the list?

This capitalizes the word but doesn't change it in the list:

```
>>> things[1].capitalize()
'Cinderella'
>>> things
['mozzarella', 'cinderella', 'salmonella']
```

If you want to change the item in the list, you need to assign it back:

```
>>> things[1] = things[1].capitalize()
>>> things
['mozzarella', 'Cinderella', 'salmonella']
```

8.6 Make the cheesy element of `things` all uppercase and then print the list.

```
>>> things[0] = things[0].upper()
>>> things
['MOZZARELLA', 'Cinderella', 'salmonella']
```

8.7 Delete the disease element from `things`, collect your Nobel Prize, and then print the list.

This would remove the element by value:

```
>>> things.remove("salmonella")
>>> things
['MOZZARELLA', 'Cinderella']
```

Because salmonella was last in the list, the following would have worked also:

```
>>> del things[-1]
```

And you could have deleted by offset from the beginning:

```
>>> del things[2]
```

8.8 Create a list called `surprise` with the elements `Groucho`, `Chico`, and `Harpo`.

```
>>> surprise = ['Groucho', 'Chico', 'Harpo']
>>> surprise
['Groucho', 'Chico', 'Harpo']
```

8.9 Lowercase the last element of the `surprise` list, reverse it, and then capitalize it.

```
>>> surprise[-1] = surprise[-1].lower()
>>> surprise[-1] = surprise[-1][::-1]
>>> surprise[-1].capitalize()
'Oprah'
```

8.10 Use a list comprehension to make a list called `even` of the even numbers in `range(10)`.

```
>>> even = [number for number in range(10) if number % 2 == 0]
>>> even
[0, 2, 4, 6, 8]
```

8.11 Let's create a jump-rope rhyme maker. You'll print a series of two-line rhymes (see Example A-6). Start with this program fragment:

```
start1 = ["fee", "fie", "foe"]
rhymes = [
    ("flop", "get a mop"),
    ("fope", "turn the rope"),
    ("fa", "get your ma"),
    ("fudge", "call the judge"),
    ("fat", "pet the cat"),
    ("fog", "walk the dog"),
    ("fun", "say we're done"),
    ]
start2 = "Someone better"
```

Do the following for each tuple (`first`, `second`) in `rhymes`.

For the first line:

- Print each string in `start1`, capitalized and followed by an exclamation point and a space.
- Print `first`, also capitalized and followed by an exclamation point.

For the second line:

- Print `start2` and a space.
- Print `second` and a period.

Example A-6. jump.py

```
start1 = ["fee", "fie", "foe"]
rhymes = [
    ("flop", "get a mop"),
    ("fope", "turn the rope"),
    ("fa", "get your ma"),
    ("fudge", "call the judge"),
    ("fat", "pet the cat"),
    ("fog", "pet the dog"),
    ("fun", "say we're done"),
    ]
start2 = "Someone better"
start1_caps = " ".join([word.capitalize() + "!" for word in start1])
for first, second in rhymes:
    print(f"{start1_caps} {first.capitalize()}!")
    print(f"{start2} {second}.")
```

Output:

```
Fee! Fie! Foe! Flop!
Someone better get a mop.
Fee! Fie! Foe! Fope!
Someone better turn the rope.
Fee! Fie! Foe! Fa!
Someone better get your ma.
Fee! Fie! Foe! Fudge!
Someone better call the judge.
Fee! Fie! Foe! Fat!
Someone better pet the cat.
Fee! Fie! Foe! Fog!
Someone better walk the dog.
Fee! Fie! Foe! Fun!
Someone better say we're done.
```

8.12 Print each list question with its correctly matching answer, in this form:

Q: *question*
A: *answer*

```
>>> questions = [
...     "We don't serve strings around here. Are you a string?",
...     "What is said on Father's Day in the forest?",
```

```
...     "What makes the sound 'Sis! Boom! Bah!'?"
...     ]
>>> answers = [
...     "An exploding sheep.",
...     "No, I'm a frayed knot.",
...     "'Pop!' goes the weasel."
...     ]
```

You could print each item in `questions` with its mate from `answers` in many ways. Let's try a tuple sandwich (tuples in a tuple) to pair them, and tuple unpacking to retrieve them for printing (Example A-7).

Example A-7. qanda.py

```
questions = [
    "We don't serve strings around here. Are you a string?",
    "What is said on Father's Day in the forest?",
    "What makes the sound 'Sis! Boom! Bah!'?"
    ]
answers = [
    "An exploding sheep.",
    "No, I'm a frayed knot.",
    "'Pop!' goes the weasel."
    ]

q_a = ( (0, 1), (1,2), (2, 0) )
for q, a in q_a:
    print("Q:", questions[q])
    print("A:", answers[a])
    print()
```

Output:

```
$ python qanda.py
Q: We don't serve strings around here. Are you a string?
A: No, I'm a frayed knot.

Q: What is said on Father's Day in the forest?
A: 'Pop!' goes the weasel.

Q: What makes the sound 'Sis! Boom! Bah!'?
A: An exploding sheep.
```

9. Dictionaries and Sets

9.1 Make an English-to-French dictionary called `e2f` and print it. Here are your starter words: `dog` is `chien`, `cat` is `chat`, and `walrus` is `morse`.

```
>>> e2f = {'dog': 'chien', 'cat': 'chat', 'walrus': 'morse'}
>>> e2f
{'cat': 'chat', 'walrus': 'morse', 'dog': 'chien'}
```

9.2 Using your three-word dictionary `e2f`, print the French word for `walrus`.

```
>>> e2f['walrus']
'morse'
```

9.3 Make a French-to-English dictionary called `f2e` from `e2f`. Use the `items()` method.

```
>>> f2e = {}
>>> for english, french in e2f.items():
    f2e[french] = english
>>> f2e
{'morse': 'walrus', 'chien': 'dog', 'chat': 'cat'}
```

9.4 Print the English equivalent of the French word `chien`.

```
>>> f2e['chien']
'dog'
```

9.5 Print the set of English words from `e2f`.

```
>>> set(e2f.keys())
{'cat', 'walrus', 'dog'}
```

9.6 Make a multilevel dictionary called `life`. Use these strings for the topmost keys: `'animals'`, `'plants'`, and `'other'`. Make the `'animals'` key refer to another dictionary with the keys `'cats'`, `'octopi'`, and `'emus'`. Make the `'cats'` key refer to a list of strings with the values `'Henri'`, `'Grumpy'`, and `'Lucy'`. Make all the other keys refer to empty dictionaries.

This is a hard one, so don't feel bad if you peeked here first.

```
>>> life = {
...     'animals': {
...         'cats': [
...             'Henri', 'Grumpy', 'Lucy'
...             ],
...         'octopi': {},
...         'emus': {}
...         },
...     'plants': {},
...     'other': {}
...     }
>>>
```

9.7 Print the top-level keys of `life`.

```
>>> print(life.keys())
dict_keys(['animals', 'other', 'plants'])
```

Python 3 includes that `dict_keys` stuff. To print them as a plain list, use this:

```
>>> print(list(life.keys()))
['animals', 'other', 'plants']
```

By the way, you can use spaces to make your code easier to read:

```
>>> print ( list ( life.keys() ) )
['animals', 'other', 'plants']
```

9.8 Print the keys for `life['animals']`.

```
>>> print(life['animals'].keys())
dict_keys(['cats', 'octopi', 'emus'])
```

9.9 Print the values for `life['animals']['cats']`.

```
>>> print(life['animals']['cats'])
['Henri', 'Grumpy', 'Lucy']
```

9.10 Use a dictionary comprehension to create the dictionary `squares`. Use `range(10)` to return the keys, and use the square of each key as its value.

```
>>> squares = {key: key*key for key in range(10)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

9.11 Use a set comprehension to create the set `odd` from the odd numbers in `range(10)`.

```
>>> odd = {number for number in range(10) if number % 2 == 1}
>>> odd
{1, 3, 5, 7, 9}
```

9.12 Use a generator comprehension to return the string `'Got '` and a number for the numbers in `range(10)`. Iterate through this by using a `for` loop.

```
>>> for thing in ('Got %s' % number for number in range(10)):
...     print(thing)
...
Got 0
Got 1
Got 2
Got 3
Got 4
Got 5
Got 6
Got 7
Got 8
Got 9
```

9.13 Use `zip()` to make a dictionary from the key tuple `('optimist', 'pessimist', 'troll')` and the values tuple `('The glass is half full', 'The glass is half empty', 'How did you get a glass?')`.

```
>>> keys = ('optimist', 'pessimist', 'troll')
>>> values = ('The glass is half full',
...     'The glass is half empty',
...     'How did you get a glass?')
>>> dict(zip(keys, values))
{'optimist': 'The glass is half full',
```

```
    'pessimist': 'The glass is half empty',
    'troll': 'How did you get a glass?'}
```

9.14 Use `zip()` to make a dictionary called `movies` that pairs the lists: `titles = ['Creature of Habit', 'Crewel Fate', 'Sharks On a Plane']` and `plots = ['A nun turns into a monster', 'A haunted yarn shop', 'Check your exits']`.

```
>>> titles = ['Creature of Habit',
...     'Crewel Fate',
...     'Sharks On a Plane']
>>> plots = ['A nun turns into a monster',
...     'A haunted yarn shop',
...     'Check your exits']
>>> movies = dict(zip(titles, plots))
>>> movies
{'Creature of Habit': 'A nun turns into a monster',
'Crewel Fate': 'A haunted yarn shop',
'Sharks On a Plane': 'Check your exits'}
>>>
```

10. Functions

10.1 Define a function called `good()` that returns the following list: `['Harry', 'Ron', 'Hermione']`.

```
>>> def good():
...     return ['Harry', 'Ron', 'Hermione']
...
>>> good()
['Harry', 'Ron', 'Hermione']
```

10.2 Define a generator function called `get_odds()` that returns the odd numbers from `range(10)`. Use a `for` loop to find and print the third value returned.

```
>>> def get_odds():
...     for number in range(1, 10, 2):
...         yield number
...
>>> count = 1
>>> for number in get_odds():
...     if count == 3:
...         print("The third odd number is", number)
```

```
...             break
...         count += 1
...
The third odd number is 5
```

10.3 Define a decorator called `test` that prints `start` when a function is called, and `end` when it finishes.

```
>>> def test(func):
...     def new_func(*args, **kwargs):
...         print('start')
...         result = func(*args, **kwargs)
...         print('end')
...         return result
...     return new_func
...
>>>
>>> @test
... def greeting():
...     print("Greetings, Earthling")
...
>>> greeting()
start
Greetings, Earthling
end
```

10.4 Define an exception called `OopsException`. Raise this exception to see what happens. Then write the code to catch this exception and print `Caught an oops`.

```
>>> class OopsException(Exception):
...     pass
...
>>> raise OopsException()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.OopsException
>>>
>>> try:
...     raise OopsException
... except OopsException:
...     print('Caught an oops')
...
Caught an oops
```

11. Objects

11.1 Make a class called `Thing` with no contents and print it. Then create an object called `example` from this class and also print it. Are the printed values the same or different?

```
>>> class Thing:
...     pass
...
>>> print(Thing)
<class '__main__.Thing'>
>>> example = Thing()
>>> print(example)
<__main__.Thing object at 0x1006f3fd0>
```

11.2 Make a new class called `Thing2` and assign the value `abc` to a class variable called `letters`. Print `letters`.

```
>>> class Thing2:
...     letters = 'abc'
...
>>> print(Thing2.letters)
abc
```

11.3 Make yet another class called, of course, `Thing3`. This time, assign the value `xyz` to an instance (object) variable called `letters`. Print `letters`.

Do you need to make an object from the class to do this?

```
>>> class Thing3:
...     def __init__(self):
...         self.letters = 'xyz'
...
```

The variable `letters` belongs to any objects made from `Thing3`, not the `Thing3` class itself:

```
>>> print(Thing3.letters)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Thing3' has no attribute 'letters'
>>> something = Thing3()
```

```
>>> print(something.letters)
xyz
```

11.4 Make a class called `Element`, with instance attributes `name`, `symbol`, and `number`. Create an object called `hydrogen` of this class with the values `Hydrogen`, `H`, and `1`.

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...
>>> hydrogen = Element('Hydrogen', 'H', 1)
```

11.5 Make a dictionary with these keys and values: `'name': 'Hydrogen', 'symbol': 'H', 'number': 1`. Then create an object called `hydrogen` from class `Element` by using this dictionary.

Start with the dictionary:

```
>>> el_dict = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1}
```

This works, although it takes a bit of typing:

```
>>> hydrogen = Element(el_dict['name'], el_dict['symbol'], el_dict['number'])
```

Let's check that it worked:

```
>>> hydrogen.name
'Hydrogen'
```

However, you can also initialize the object directly from the dictionary, because its key names match the arguments to `__init__` (refer to Chapter 10 for a discussion of keyword arguments):

```
>>> hydrogen = Element(**el_dict)
>>> hydrogen.name
'Hydrogen'
```

11.6 For the `Element` class, define a method called `dump()` that prints the values of the object's attributes (`name`, `symbol`, and `number`). Create the `hydrogen` object from this new definition and use `dump()` to print its attributes.

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...     def dump(self):
...         print('name=%s, symbol=%s, number=%s' %
...             (self.name, self.symbol, self.number))
...
>>> hydrogen = Element(**el_dict)
>>> hydrogen.dump()
name=Hydrogen, symbol=H, number=1
```

11.7 Call `print(hydrogen)`. In the definition of `Element`, change the name of the method `dump` to `__str__`, create a new `hydrogen` object, and call `print(hydrogen)` again.

```
>>> print(hydrogen)
<__main__.Element object at 0x1006f5310>
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...     def __str__(self):
...         return ('name=%s, symbol=%s, number=%s' %
...             (self.name, self.symbol, self.number))
...
>>> hydrogen = Element(**el_dict)
>>> print(hydrogen)
name=Hydrogen, symbol=H, number=1
```

`__str__()` is one of Python's *magic methods*. The `print` function calls an object's `__str__()` method to get its string representation. If it doesn't have a `__str__()` method, it gets the default method from its parent `Object` class, which returns a string like `<__main__.Element object at 0x1006f5310>`.

11.8 Modify `Element` to make the attributes `name`, `symbol`, and `number` private. Define a getter property for each to return its value.

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.__name = name
...         self.__symbol = symbol
```

```
...         self.__number = number
...     @property
...     def name(self):
...         return self.__name
...     @property
...     def symbol(self):
...         return self.__symbol
...     @property
...     def number(self):
...         return self.__number
...
>>> hydrogen = Element('Hydrogen', 'H', 1)
>>> hydrogen.name
'Hydrogen'
>>> hydrogen.symbol
'H'
>>> hydrogen.number
1
```

11.9 Define three classes: `Bear`, `Rabbit`, and `Octothorpe`. For each, define only one method: `eats()`. This should return `berries` (`Bear`), `clover` (`Rabbit`), and `campers` (`Octothorpe`). Create one object from each and print what it eats.

```
>>> class Bear:
...     def eats(self):
...         return 'berries'
...
>>> class Rabbit:
...     def eats(self):
...         return 'clover'
...
>>> class Octothorpe:
...     def eats(self):
...         return 'campers'
...
>>> b = Bear()
>>> r = Rabbit()
>>> o = Octothorpe()
>>> print(b.eats())
berries
>>> print(r.eats())
clover
>>> print(o.eats())
campers
```

11.10 Define these classes: `Laser`, `Claw`, and `SmartPhone`. Each has only one method: `does()`. This returns `disintegrate` (`Laser`), `crush` (`Claw`), or `ring` (`SmartPhone`). Then define the class `Robot` that has one instance (object) of each of these. Define a `does()` method for the `Robot` that prints what its component objects do.

```
>>> class Laser:
...     def does(self):
...         return 'disintegrate'
...
>>> class Claw:
...     def does(self):
...         return 'crush'
...
>>> class SmartPhone:
...     def does(self):
...         return 'ring'
...
>>> class Robot:
...     def __init__(self):
...         self.laser = Laser()
...         self.claw = Claw()
...         self.smartphone = SmartPhone()
...     def does(self):
...         return '''I have many attachments:
... My laser, to %s.
... My claw, to %s.
... My smartphone, to %s.''' % (
...     self.laser.does(),
...     self.claw.does(),
...     self.smartphone.does() )
...
>>> robbie = Robot()
>>> print( robbie.does() )
I have many attachments:
My laser, to disintegrate.
My claw, to crush.
My smartphone, to ring.
```

12. Modules and Packages

12.1 Make a file called *zoo.py*. In it, define a function called `hours()` that prints the string `Open 9-5 daily`. Then use the interactive interpreter to import the `zoo` module and call its `hours()` function.

Example A-8 shows *zoo.py*.

Example A-8. zoo.py

```
def hours():
    print('Open 9-5 daily')
```

And now, let's import it interactively:

```
>>> import zoo
>>> zoo.hours()
Open 9-5 daily
```

12.2 In the interactive interpreter, import the `zoo` module as `menagerie` and call its `hours()` function.

```
>>> import zoo as menagerie
>>> menagerie.hours()
Open 9-5 daily
```

12.3 Staying in the interpreter, import the `hours()` function from `zoo` directly and call it.

```
>>> from zoo import hours
>>> hours()
Open 9-5 daily
```

12.4 Import the `hours()` function as `info` and call it.

```
>>> from zoo import hours as info
>>> info()
Open 9-5 daily
```

12.5 Make a dictionary called `plain` with the key-value pairs *a*: 1, *b*: 2, and *c*: 3, and then print it.

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

12.6 Make an `OrderedDict` called `fancy` from the same pairs and print it. Did it print in the same order as `plain`?

```
>>> from collections import OrderedDict
>>> fancy = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> fancy
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

12.7 Make a `defaultdict` called `dict_of_lists` and pass it the argument `list`. Make the list `dict_of_lists['a']` and append the value `something for a` to it in one assignment. Print `dict_of_lists['a']`.

```
>>> from collections import defaultdict
>>> dict_of_lists = defaultdict(list)
>>> dict_of_lists['a'].append('something for a')
>>> dict_of_lists['a']
['something for a']
```

13. Development Environment

13.1 Install `pip` if you don't have it. Make `pip` install the pyspellchecker package.

```
$ pip install pyspellchecker
```

13.2 Read the pyspellchecker docs at the PyPI and Read the Docs websites. Write a small script that takes a list of words, reports whether any are misspelled, and reports the likely true spelling. Note that more than 10 languages are supported. See Example A-9.

Example A-9. check.py

```
import spellchecker as sp

ch = sp.SpellChecker()
words = ('mispell', 'phantom', 'tumeric', 'dilatation')
wrong = ch.unknown(words)
for word in words:
    print(word)
```

```
    if word in wrong:
        print(' '*4, ch.correction(word), ch.candidates(word))
```

```
$ python check.py
mispell
     misspell {misspell, ispell}
phantom
tumeric
     numeric {numeric, turmeric}
dilatation
```

13.3 (Optional) Install uv. Use it to check the correctness of the code you wrote for 13.2.

13.4 (Optional) Install PyCharm and VS Code. Try them out.

14. Type Hints and Documentation

14.1 Write a function called `pointless()` that accepts a dict of *`string:int`* elements, capitalizes the key, adds 1 to the value, and prints both. Add complete type hints. See Example A-10.

Example A-10. hints.py

```
def pointless(data: dict[str, int]) -> None:
    for key, val in data.items():
        print(key.capitalize(), val + 1)

data = {'hawaii':49, 'carbon': 13}
pointless(data)
```

```
$ python hints.py
Hawaii 50
Carbon 14
```

15. Testing

15.1 Find out what's wrong with the code in Example A-11. Use Pylint, Pyflakes, and Ruff (install them if needed).

Example A-11. errors.py

```
import math

print("square root of 3 is", sqrt(3))
```

```
$ pylint errors.py
* Module errors
errors.py:1:0: C0114: Missing module docstring (missing-module-docstring)
errors.py:3:29: E0602: Undefined variable sqrt (undefined-variable)
errors.py:1:0: W0611: Unused import math (unused-import)

-----------------------------------
Your code has been rated at 0.00/10
```

```
$ pyflakes errors.py
errors.py:1:1: math imported but unused
errors.py:3:30: undefined name sqrt
```

```
$ ruff check errors.py
errors.py:1:8: F401 [*] `math` imported but unused
  |
1 | import math
  |        ^^ F401
2 |
3 | print("square root of 3 is", sqrt(3))
  |
  = help: Remove unused import: `math`

errors.py:3:30: F821 Undefined name `sqrt`
  |
1 | import math
2 |
3 | print("square root of 3 is", sqrt(3))
  |                              ^^ F821
  |

Found 2 errors.
[*] 1 fixable with the `--fix` option.
```

15.2 Write a Python file that contains the following (Example A-12):

- A function that adds two integers and returns their sum
- Some pytest tests of this function
- A fixture that returns the current date and time when the file was called

Example A-12. test_arino.py

```
import pytest
import datetime

def add_integers(a, b):
    return a + b

@pytest.fixture(scope="module")
def curtime():
    return datetime.datetime.now(datetime.UTC)

def test_zeroes(curtime):
    print(f"{curtime=}")
    assert(add_integers(0, 0) == 0)

def test_positive_ints(curtime):
    print(f"{curtime=}")
    assert(add_integers(5, 6) == 11)

def test_negative_ints(curtime):
    print(f"{curtime=}")
    assert(add_integers(-1, -1) == -2)
```

```
$ pytest -sv test_arino.py
========================== test session starts ============================
...
collected 3 items

test_arino.py::test_zeroes curtime =  2025-06-01 18:46:04.525185+00:00
PASSED
test_arino.py::test_positive_ints curtime =  2025-06-01 18:46:04.525185+00:00
PASSED
test_arino.py::test_negative_ints curtime =  2025-06-01 18:46:04.525185+00:00
PASSED

============================ 3 passed in 0.20s ============================
```

16. Debugging

16.1 Use the f-string debug print feature to print these expressions:

- Assign 5 to x, then print double it if x is odd; otherwise, print 2.
- Print x using 20 spaces of width, center adjusted, with dots filling the left and right spaces.

```
>>> x = 5
>>> print(f"{2*x if x%2 == 1 else 2}")
10
>>> print(f"{x:.^20}")
.........5.........
```

17. Text Data

17.1 Create a Unicode string called `mystery` and assign it the value `\U0001f984`. Print `mystery` and its Unicode name. Then assign `\U0001f4a9` to `mystery2` and do the same. (Your system font may or may not display an image for both.)

```
>>> import unicodedata
>>> mystery = '\U0001f984'
>>> print(mystery)
🦄
>>> unicodedata.name(mystery)
'UNICORN FACE'
>>> mystery2 = '\U0001f4a9'
>>> print(mystery2)
💩
>>> unicodedata.name(mystery2)
'PILE OF POO'
```

Hooray and oh dear. I thought unicorns made rainbows.

17.2 Encode `mystery`, this time using UTF-8, into the `bytes` variable `pop_bytes`. Print `pop_bytes`.

```
>>> pop_bytes = mystery.encode('utf-8')
>>> pop_bytes
b'\xf0\x9f\xa6\x84'
```

17.3 Using UTF-8, decode `pop_bytes` into the string variable `pop_string`. Print `pop_string`. Is `pop_string` equal to `mystery`?

```
>>> pop_string = pop_bytes.decode('utf-8')
>>> print(pop_string)
🦄
>>> pop_string == mystery
True
```

17.4 When you're working with text, regular expressions come in handy. We'll apply them in numerous ways to our featured text sample. It's a poem titled "Ode on the Mammoth Cheese," written by James McIntyre in 1866 in homage to a seven-thousand-pound cheese that was crafted in Ontario and sent on an international tour. If you'd rather not type all of it, use your favorite search engine and cut and paste the words into your Python program, or just grab it from Project Gutenberg. Call the text string `mammoth`.

```
>>> mammoth = '''
... We have seen thee, queen of cheese,
... Lying quietly at your ease,
... Gently fanned by evening breeze,
... Thy fair form no flies dare seize.
...
... All gaily dressed soon you'll go
... To the great Provincial show,
... To be admired by many a beau
... In the city of Toronto.
...
... Cows numerous as a swarm of bees,
... Or as the leaves upon the trees,
... It did require to make thee please,
... And stand unrivalled, queen of cheese.
...
... May you not receive a scar as
... We have heard that Mr. Harris
... Intends to send you off as far as
... The great world's show at Paris.
...
... Of the youth beware of these,
... For some of them might rudely squeeze
... And bite your cheek, then songs or glees
... We could not sing, oh! queen of cheese.
...
... We'rt thou suspended from balloon,
... You'd cast a shade even at noon,
... Folks would think it was the moon
... About to fall and crush them soon.
... '''
```

17.5 Import the `re` module to use Python's regular expression functions. Use `re.findall()` to print all the words that begin with `c`.

We'll define the variable `pat` for the pattern and then search for it in `mammoth`:

```
>>> import re
>>> pat = r'\bc\w*'
>>> re.findall(pat, mammoth)
['cheese', 'city', 'cheese', 'cheek', 'could', 'cheese', 'cast', 'crush']
```

The `\b` means to begin at a boundary between a word and a nonword. Use this to specify either the beginning or end of a word. The literal `c` is the first letter of the words we're looking for. The `\w` means any *word character*, which includes letters, digits, and underscores (_). The `*` means *zero or more* of these word characters. Together, this finds words that begin with `c`, including `'c'` itself. If you didn't use a raw string (with an `r` right before the starting quote), Python would interpret `\b` as a backspace and the search would mysteriously fail:

```
>>> pat = '\bc\w*'
>>> re.findall(pat, mammoth)
[]
```

17.6 Find all four-letter words that begin with `c`.

```
>>> pat = r'\bc\w{3}\b'
>>> re.findall(pat, mammoth)
['city', 'cast']
```

You need that final `\b` to indicate the end of the word. Otherwise, you'll get the first four letters of all words that begin with `c` and have at least four letters:

```
>>> pat = r'\bc\w{3}'
>>> re.findall(pat, mammoth)
['chee', 'city', 'chee', 'chee', 'coul', 'chee', 'cast', 'crus']
```

17.7 Find all the words that end with `r`.

This is a little tricky. We get the right result for words that end with `r`:

```
>>> pat = r'\b\w*r\b'
>>> re.findall(pat, mammoth)
['your', 'fair', 'Or', 'scar', 'Mr', 'far', 'For', 'your', 'or']
```

However, the results aren't so good for words that end with `l`:

```
>>> pat = r'\b\w*l\b'
>>> re.findall(pat, mammoth)
['All', 'll', 'Provincial', 'fall']
```

But what's that `ll` doing there? The `\w` pattern matches only letters, numbers, and underscores—not ASCII apostrophes. As a result, the code grabs the final `ll` from `you'll`. We can handle this by adding an apostrophe to the set of characters to match. Our first try fails:

```
>>> pat = r'\b[\w']*l\b'
  File "<stdin>", line 1
    pat = r'\b[\w']*l\b'
```

Python points to the vicinity of the error, but it might take a while to see that the mistake was that the pattern string is surrounded by the same apostrophe/quote character. One way to solve this is to escape it with a backslash:

```
>>> pat = r'\b[\w\']*l\b'
>>> re.findall(pat, mammoth)
['All', "you'll", 'Provincial', 'fall']
```

Another way is to surround the pattern string with double quotes:

```
>>> pat = r"\b[\w']*l\b"
>>> re.findall(pat, mammoth)
['All', "you'll", 'Provincial', 'fall']
```

17.8 Find all the words that contain exactly three vowels in a row.

Begin with a word boundary, any number of *word* characters, three vowels, and then any nonvowel characters to the end of the word:

```
>>> pat = r'\b[^aeiou]*[aeiou]{3}[^aeiou]*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'beau\nIn', 'queen', 'squeeze', 'queen']
```

This looks right, except for that `beau\nIn` string. We searched `mammoth` as a single multiline string. Our `[^aeiou]` matches any nonvowels, including `\n` (line feed, which marks the end of a text line). We need to add one more detail to the ignore set: `\s` matches any space characters, including `\n`:

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou\s]\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'queen', 'squeeze', 'queen']
```

We didn't find `beau` this time, so we need one more tweak to the pattern: match any number (even zero) of nonvowels after the three vowels. Our previous pattern always matched one nonvowel:

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou\s]*\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'beau', 'queen', 'squeeze', 'queen']
```

What does all this show? Among other things, that regular expressions can do a lot, but they can be very tricky to get right.

18. Binary Data

18.1 Use `unhexlify()` to convert this hex string (combined from two strings to fit on a page) to a `bytes` variable called `gif`:

```
'4749463839610100010080000000000ffffff21f9' +
'0401000000002c000000000100010000020144003b'
```

```
>>> import binascii
>>> hex_str = '4749463839610100010080000000000ffffff21f9' + \
...     '0401000000002c000000000100010000020144003b'
>>> gif = binascii.unhexlify(hex_str)
>>> len(gif)
42
```

18.2 The bytes in `gif` define a 1-pixel transparent GIF file, one of the most common graphics file formats. A legal GIF starts with the string `GIF89a`. Does `gif` match this?

```
>>> gif[:6] == b'GIF89a'
True
```

Notice that we need to use a `b` to define a byte string rather than a Unicode character string. You can compare bytes with bytes, but you cannot compare bytes with strings:

```
>>> gif[:6] == 'GIF89a'
False
>>> type(gif)
<class 'bytes'>
>>> type('GIF89a')
<class 'str'>
>>> type(b'GIF89a')
<class 'bytes'>
```

18.3 The pixel width of a GIF is a 16-bit little-endian integer starting at byte offset 6, and the height is the same size, starting at offset 8. Extract and print these values for `gif`. Are they both `1`?

```
>>> import struct
>>> width, height = struct.unpack('<HH', gif[6:10])
>>> width, height
(1, 1)
```

19. Dates and Times

19.1 Write the current date as a string to the text file *today.txt*.

```
>>> from datetime import date
>>> now = date.today()
>>> now_str = now.isoformat()
>>> with open('today.txt', 'wt') as output:
...     print(now_str, file=output)
>>>
```

When I ran this, here's what I got in *today.txt*:

```
2025-04-23
```

Instead of `print`, you could have also said something like `output.write(now_str)`. Using `print` adds the final newline.

19.2 Read the text file *today.txt* into the string `today_string`.

```
>>> with open('today.txt', 'rt') as input:
...     today_string = input.read()
...
>>> today_string
'2025-04-23\n'
```

19.3 Parse the date from `today_string`.

```
>>> fmt = '%Y-%m-%d\n'
>>> datetime.strptime(today_string, fmt)
datetime.datetime(2019, 7, 23, 0, 0)
```

If you wrote that final newline to the file, you need to match it in the format string.

19.4 Create a date object of your day of birth.

Let's say that you were born on August 14, 1982:

```
>>> my_day = date(1982, 8, 14)
>>> my_day
datetime.date(1982, 8, 14)
```

19.5 What day of the week was your day of birth?

```
>>> my_day.weekday()
5
>>> my_day.isoweekday()
6
```

With `weekday()`, Monday is 0 and Sunday is 6. With `isoweekday()`, Monday is 1 and Sunday is 7. Therefore, this date was a Saturday.

19.6 When will you be (or when were you) 10,000 days old?

```
>>> from datetime import timedelta
>>> party_day = my_day + timedelta(days=10000)
>>> party_day
datetime.date(2009, 12, 30)
```

If August 14, 1982 was your birthday, you probably missed an excuse for a party.

20. Files

20.1 List the files in your current directory.

If your current directory is *ohmy* and contains three files named after animals, it might look like this:

```
>>> import os
>>> os.listdir('.')
['bears', 'lions', 'tigers']
```

20.2 List the files in your parent directory.

If your parent directory contains two files plus the current *ohmy* directory, it might look like this:

```
>>> import os
>>> os.listdir('..')
['ohmy', 'paws', 'whiskers']
```

20.3 Assign the string `This is a test of the emergency text system` to the variable `test1`, and write `test1` to a file called *test.txt*.

```
>>> test1 = 'This is a test of the emergency text system'
>>> len(test1)
43
```

Here's how to do it by using `open()`, `write()`, and `close()`:

```
>>> outfile = open('test.txt', 'wt')
>>> outfile.write(test1)
43
>>> outfile.close()
```

Or, you can use `with` and avoid calling `close()` (Python does it for you):

```
>>> with open('test.txt', 'wt') as outfile:
...     outfile.write(test1)
...
43
```

20.4 Open the file *test.txt* and read its contents into the string `test2`. Are `test1` and `test2` the same?

```
>>> with open('test.txt', 'rt') as infile:
...     test2 = infile.read()
...
>>> len(test2)
43
>>> test1 == test2
True
```

21. Data in Time: Concurrency

21.1 Use `multiprocessing` to create three separate processes, as shown in Example A-13. Make each one wait a random number of seconds from 0 to 1, print the current time, and then exit.

Example A-13. multi_times.py

```
import multiprocessing

def now(seconds):
    from datetime import datetime
    from time import sleep
    sleep(seconds)
    print('wait', seconds, 'seconds, time is', datetime.utcnow())

if __name__ == '__main__':
    import random
    for n in range(3):
        seconds = random.random()
        proc = multiprocessing.Process(target=now, args=(seconds,))
        proc.start()
```

```
$ python multi_times.py
wait 0.10720361113059229 seconds, time is 2019-07-24 00:19:23.951594
wait 0.5825144002370065 seconds, time is 2019-07-24 00:19:24.425047
wait 0.6647690569029477 seconds, time is 2019-07-24 00:19:24.509995
```

22. Data in Space: Networks

22.1 Use a plain `socket` to implement a current-time service. When a client sends the string `time` to the server, return the current date and time as an ISO string.

Example A-14 shows one way to write the server, *udp_time_server.py*. Note that if you use `datetime.datetime.utcnow()`, for the current time in UTC, you'll get a `DeprecationWarning`. This will go away in a future version of Python; the recommended replacement is `datetime.datetime.now(datetime.UTC)`.

Example A-14. udp_time_server.py

```
import datetime
import socket

address = ('localhost', 6789)
max_size = 4096
```

```
print('Starting the server at', datetime.datetime.now(datetime.UTC))
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(address)
while True:
    data, client_addr = server.recvfrom(max_size)
    if data == b'time':
        now = str(datetime.datetime.now(datetime.UTC))
        data = now.encode('utf-8')
        server.sendto(data, client_addr)
        print('Server sent', data)
server.close()
```

And create the client, *udp_time_client.py*, as shown in Example A-15.

Example A-15. udp_time_client.py

```
import socket
from datetime import datetime
from time import sleep

address    = ('localhost', 6789)
max_size   = 4096

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    sleep(5)
    client.sendto(b'time', address)
    data, server_addr = client.recvfrom(max_size)
    print('Client read', data)
client.close()
```

I put in a `sleep(5)` call at the top of the client loop to make the data exchange less supersonic. Start the server in one window:

```
$ python udp_time_server.py
Starting the server at 2025-04-23 23:14:12.095702
Waiting for a client to call.
```

Start the client in another window:

```
$ python udp_time_client.py
Starting the client at 2025-04-23 23:14:39.915397
```

After five seconds, you'll start getting output in both windows. Here are the first three lines from the server:

```
Server sent b'2025-04-24 04:14:44.924383'
Server sent b'2025-04-24 04:14:49.929807'
Server sent b'2025-04-24 04:14:54.931491'
```

And here are the first three from the client:

```
Client read b'2025-04-24 04:14:44.924383'
Client read b'2025-04-24 04:14:49.929807'
Client read b'2025-04-24 04:14:54.931491'
```

Both of these programs run forever, so you'll need to cancel them manually.

22.2. Use ZeroMQ `REQ` and `REP` sockets to do the same thing. See Examples A-16 and A-17.

Example A-16. zmq_time_server.py

```
import zmq
import datetime

host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
now = datetime.datetime.now(datetime.UTC)
print('Server started at', now)
while True:
    #  Wait for next request from client
    message = server.recv()
    if message == b'time':
        now = datetime.datetime.now(datetime.UTC)
        reply = str(now)
        server.send(bytes(reply, 'utf-8'))
        print('Server sent', reply)
```

Example A-17. zmq_time_client.py

```
import zmq
import datetime
from time import sleep

host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
now = datetime.datetime.now(datetime.UTC)
print('Client started at', now)
while True:
    sleep(5)
    request = b'time'
    client.send(request)
```

```
    reply = client.recv()
    print("Client received %s" % reply)
```

With plain sockets, you need to start the server first. With ZeroMQ, you can start either the server or client first:

```
$ python zmq_time_server.py
Server started at 2025-04-24 04:32:31.014781+00:00

$ python zmq_time_client.py
Client started at 2025-04-24 04:32:33.877329+00:00
```

After 15 seconds or so, you should have some lines from the server:

```
Server sent 2025-04-24 04:32:38.882717+00:00
Server sent 2025-04-24 04:32:43.888701+00:00
Server sent 2025-04-24 04:32:48.894584+00:00
```

Here's what you should see from the client:

```
Client received b'2025-04-24 04:32:38.882717+00:00'
Client received b'2025-04-24 04:32:43.888701+00:00'
Client received b'2025-04-24 04:32:48.894584+00:00'
```

22.3. Try the same with XML-RPC.

Example A-18 has the server code.

Example A-18. xmlrpc_time_server.py

```
from xmlrpc.server import SimpleXMLRPCServer

def now():
    import datetime
    data = str(datetime.datetime.now(datetime.UTC))
    print('Server sent', data)
    return data

server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(now, "now")
server.serve_forever()
```

And Example A-19 has the corresponding client code.

Example A-19. xmlrpc_time_client.py

```
import xmlrpc.client
from time import sleep

proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
while True:
    sleep(5)
```

```
    data = proxy.now()
    print('Client received', data)
```

Start the server:

```
$ python xmlrpc_time_server.py
```

Start the client:

```
$ python xmlrpc_time_client.py
```

Wait 15 seconds or so. Here are the first three lines of server output:

```
Server sent 2025-04-24 04:39:14.842428+00:00
127.0.0.1 - - [23/Apr/2025 23:39:14] "POST / HTTP/1.1" 200 -
Server sent 2025-04-24 04:39:19.847258+00:00
127.0.0.1 - - [23/Apr/2025 23:39:19] "POST / HTTP/1.1" 200 -
Server sent 2025-04-24 04:39:24.851759+00:00
127.0.0.1 - - [23/Apr/2025 23:39:24] "POST / HTTP/1.1" 200 -
```

And here are the first three lines from the client:

```
Client received 2025-04-24 04:39:14.842428+00:00
Client received 2025-04-24 04:39:19.847258+00:00
Client received 2025-04-24 04:39:24.851759+00:00
```

22.4 You may have seen the classic *I Love Lucy* television episode in which Lucy and Ethel work in a chocolate factory. The duo falls behind as the conveyor belt that supplies the confections for them to process begins operating at an ever-faster rate. Write a simulation that pushes different types of chocolates to a Redis list, and Lucy is a client doing Redis "blocking pops" of this list. She needs 0.5 seconds to handle a piece of chocolate. Print the time and type of each chocolate as Lucy gets it, as well as the number that remain to be handled.

The *redis_choc_supply.py* file supplies the infinite treats, as shown in Example A-20.

Example A-20. redis_choc_supply.py

```
import redis
import random
from time import sleep

conn = redis.Redis()
varieties = ['truffle', 'cherry', 'caramel', 'nougat']
conveyor = 'chocolates'
while True:
    seconds = random.random()
    sleep(seconds)
    piece = random.choice(varieties)
    conn.rpush(conveyor, piece)
```

The *redis_lucy.py* file might look like Example A-21.

Example A-21. redis_lucy.py

```
import redis
from datetime import datetime
from time import sleep

conn = redis.Redis()
timeout = 10
conveyor = 'chocolates'
while True:
    sleep(0.5)
    msg = conn.blpop(conveyor, timeout)
    remaining = conn.llen(conveyor)
    if msg:
        piece = msg[1]
        print('Lucy got a', piece, 'at', datetime.utcnow(),
        ', only', remaining, 'left')
```

Start the Redis server, then start the two preceding scripts in either order. Because Lucy takes a half second to handle each, and they're being produced every half second on average, it's a race to keep up. The more of a head start that you give to the conveyor belt, the harder you make Lucy's life:

```
$ python redis_choc_supply.py &

$ python redis_lucy.py
Lucy got a b'truffle' at 2025-06-04 21:58:25.088134 , only 22 left
Lucy got a b'nougat' at 2025-06-04 21:58:25.589885 , only 22 left
Lucy got a b'cherry' at 2025-06-04 21:58:26.092401 , only 21 left
Lucy got a b'cherry' at 2025-06-04 21:58:26.594963 , only 21 left
Lucy got a b'nougat' at 2025-06-04 21:58:27.097996 , only 20 left
Lucy got a b'caramel' at 2025-06-04 21:58:27.600008 , only 21 left
Lucy got a b'truffle' at 2025-06-04 21:58:28.102135 , only 22 left
Lucy got a b'cherry' at 2025-06-04 21:58:28.604621 , only 21 left
Lucy got a b'nougat' at 2025-06-04 21:58:29.107136 , only 22 left
Lucy got a b'cherry' at 2025-06-04 21:58:29.609579 , only 22 left
Lucy got a b'cherry' at 2025-06-04 21:58:30.111380 , only 23 left
Lucy got a b'truffle' at 2025-06-04 21:58:30.613568 , only 23 left
Lucy got a b'caramel' at 2025-06-04 21:58:31.115780 , only 24 left
Lucy got a b'cherry' at 2025-06-04 21:58:31.618605 , only 23 left
Lucy got a b'truffle' at 2025-06-04 21:58:32.121005 , only 23 left
Lucy got a b'cherry' at 2025-06-04 21:58:32.623156 , only 23 left
Lucy got a b'cherry' at 2025-06-04 21:58:33.125025 , only 23 left
Lucy got a b'cherry' at 2025-06-04 21:58:33.626977 , only 23 left
Lucy got a b'cherry' at 2025-06-04 21:58:34.128611 , only 23 left
```

Poor Lucy.

23. Data in a Box: Persistent Storage

23.1 Save the following text lines to a file called *books.csv* (notice that if the fields are separated by commas, you need to surround any field containing a comma with quotes):

```
author,book
J R R Tolkien,The Hobbit
Lynne Truss,"Eats, Shoots & Leaves"
```

```
>>> text = '''author,book
... J R R Tolkien,The Hobbit
... Lynne Truss,"Eats, Shoots & Leaves"
... '''
>>> with open('test.csv', 'wt') as outfile:
...     outfile.write(text)
...
73
```

23.2 Use the `csv` module and its `DictReader()` method to read *books.csv* to the variable `books`. Print the values in `books`. Did `DictReader()` handle the quotes and commas in the second book's title?

```
>>> with open('books.csv', 'rt') as infile:
...     books = csv.DictReader(infile)
...     for book in books:
...         print(book)
...
{'book': 'The Hobbit', 'author': 'J R R Tolkien'}
{'book': 'Eats, Shoots & Leaves', 'author': 'Lynne Truss'}
```

23.3 Create a CSV file called *books2.csv* by using these lines:

```
title,author,year
The Weirdstone of Brisingamen,Alan Garner,1960
Perdido Street Station,China Miéville,2000
Thud!,Terry Pratchett,2005
The Spellman Files,Lisa Lutz,2007
Small Gods,Terry Pratchett,1992
```

```
>>> text = '''title,author,year
... The Weirdstone of Brisingamen,Alan Garner,1960
... Perdido Street Station,China Miéville,2000
... Thud!,Terry Pratchett,2005
... The Spellman Files,Lisa Lutz,2007
... Small Gods,Terry Pratchett,1992
... '''
>>> with open('books2.csv', 'wt') as outfile:
...     outfile.write(text)
...
201
```

23.4 Use the `sqlite3` module to create a SQLite database called *books.db* and a table called `books` with these fields: `title` (text), `author` (text), and `year` (integer).

```
>>> import sqlite3
>>> db = sqlite3.connect('books.db')
>>> curs = db.cursor()
>>> curs.execute('''create table book (title text, author text, year int)''')
<sqlite3.Cursor object at 0x1006e3b90>
>>> db.commit()
```

23.5 Read *books2.csv* and insert its data into the `book` table.

```
>>> import csv
>>> import sqlite3
>>> ins_str = 'insert into book values(?, ?, ?)'
>>> with open('books.csv', 'rt') as infile:
...   books = csv.DictReader(infile)
...   for book in books:
...     curs.execute(ins_str, (book['title'], book['author'], book['year']))
...
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
>>> db.commit()
```

23.6 Select and print the `title` column from the book table in alphabetical order.

```
>>> sql = 'select title from book order by title asc'
>>> for row in db.execute(sql):
...     print(row)
...
('Perdido Street Station',)
('Small Gods',)
('The Spellman Files',)
('The Weirdstone of Brisingamen',)
('Thud!',)
```

If you just wanted to print the `title` value without that tuple stuff (parentheses and comma), try this:

```
>>> for row in db.execute(sql):
...     print(row[0])
...
Perdido Street Station
Small Gods
The Spellman Files
The Weirdstone of Brisingamen
Thud!
```

If you want to ignore the initial The in titles, you need a little extra SQL fairy dust:

```
>>> sql = '''select title from book order by
... case when (title like "The %")
... then substr(title, 5)
... else title end'''
>>> for row in db.execute(sql):
...     print(row[0])
...
Perdido Street Station
Small Gods
The Spellman Files
Thud!
The Weirdstone of Brisingamen
```

23.7 Select and print all columns from the book table in order of publication.

```
>>> for row in db.execute('select * from book order by year'):
...     print(row)
...
('The Weirdstone of Brisingamen', 'Alan Garner', 1960)
('Small Gods', 'Terry Pratchett', 1992)
('Perdido Street Station', 'China Miéville', 2000)
('Thud!', 'Terry Pratchett', 2005)
('The Spellman Files', 'Lisa Lutz', 2007)
```

To print all the fields in each row, separate the fields with a comma and space:

```
>>> for row in db.execute('select * from book order by year'):
...     print(*row, sep=', ')
...
The Weirdstone of Brisingamen, Alan Garner, 1960
Small Gods, Terry Pratchett, 1992
Perdido Street Station, China Miéville, 2000
Thud!, Terry Pratchett, 2005
The Spellman Files, Lisa Lutz, 2007
```

23.8 Use the SQLAlchemy module to connect to the sqlite3 database *books.db* that you just made in Exercise 23.4. As in 23.6, select and print the `title` column from the book table in alphabetical order.

```
>>> import sqlalchemy
>>> conn = sqlalchemy.create_engine('sqlite:///books.db')
>>> sql = 'select title from book order by title asc'
>>> rows = conn.execute(sql)
>>> for row in rows:
...     print(row)
...
('Perdido Street Station',)
('Small Gods',)
('The Spellman Files',)
('The Weirdstone of Brisingamen',)
('Thud!',)
```

23.9 Install the Redis server and the Python redis library (`pip install redis`) on your machine. Create a Redis hash called `test` with the fields `count` (`1`) and `name` (`Fester Bestertester`). Print all the fields for `test`.

```
>>> import redis
>>> conn = redis.Redis()
>>> conn.delete('test')
1
>>> conn.hmset('test', {'count': 1, 'name': 'Fester Bestertester'})
True
>>> conn.hgetall('test')
{b'name': b'Fester Bestertester', b'count': b'1'}
```

23.10 Increment the `count` field of `test` and print it.

```
>>> conn.hincrby('test', 'count', 3)
4
>>> conn.hget('test', 'count')
b'4'
```

24. The Web

24.1 If you haven't installed Flask yet, do so now. This will also install Werkzeug, Jinja2, and possibly other packages.

24.2 Build a skeleton website, using Flask's debug/reload development web server. Ensure that the server starts up for hostname `localhost` on default port `5000`. If your machine is already using port 5000 for something else, use another port number.

Example A-22 shows *flask1.py*.

Example A-22. flask1.py

```
from flask import Flask

app = Flask(__name__)

app.run(port=5000, debug=True)
```

Gentlemen, start your engines:

```
$ python flask1.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

24.3 Add a `home()` function to handle requests for the home page. Set it up to return the string `It's alive!`.

What should we call this one, *flask2.py* (Example A-23)?

Example A-23. flask2.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
```

```
def home():
    return "It's alive!"

app.run(debug=True)
```

Start the server:

```
$ python flask2.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

Finally, access the home page via a browser, command-line HTTP program such as `curl` or `wget`, or even `telnet`:

```
$ curl http://localhost:5000/
It's alive!
```

24.4 Create a Jinja2 template file called *home.html* with the following contents:

```
<html>
<head>
<title>It's alive!</title>
<body>
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
</body>
</html>
```

Make a directory called *templates* and create the file *home.html* with the contents just shown. If your Flask server is still running from the previous examples, it will detect the new content and restart itself.

24.5 Modify your server's `home()` function to use the *home.html* template. Provide it with three `GET` parameters: `thing`, `height`, and `color`.

Example A-24 shows *flask3.py*.

Example A-24. flask3.py

```
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/')
def home():
    thing = request.values.get('thing')
    height = request.values.get('height')
    color = request.values.get('color')
    return render_template('home.html',
        thing=thing, height=height, color=color)
```

```
app.run(debug=True)
```

Go to this address in your web client:

```
http://localhost:5000/?thing=Octothorpe&height=7&color=green
```

You should see the following:

```
I'm of course referring to Octothorpe, which is 7 feet tall and green.
```

25. Data Science

25.1 Install DuckDB, and re-create the zoo database from "SQLite" on page 429, as shown in Example A-25.

Example A-25. duck.py

```
import duckdb

conn = duckdb.connect(database = ":memory:")
conn.execute('''create table zoo
   (critter varchar(20) primary key,
   count int,
   damages float)''')
ins = 'insert into zoo (critter, count, damages) values(?, ?, ?)'
conn.execute(ins, ('duck', 5, 0.0))
conn.execute(ins, ('bear', 2, 1000.0))
conn.execute(ins, ('weasel', 1, 2000.0))
conn.execute('select * from zoo')
print(conn.fetchall())
conn.execute('select * from zoo order by count')
print(conn.fetchall())
conn.execute('select * from zoo order by count desc')
print(conn.fetchall())
conn.execute('''select * from zoo where
    damages = (select max(damages) from zoo)''')
print(conn.fetchall())
```

```
$ python duck.py
[(duck, 5, 0.0), (bear, 2, 1000.0), (weasel, 1, 2000.0)]
[(weasel, 1, 2000.0), (bear, 2, 1000.0), (duck, 5, 0.0)]
[(duck, 5, 0.0), (bear, 2, 1000.0), (weasel, 1, 2000.0)]
[(weasel, 1, 2000.0)]
```

26. AI

In "Working Examples With Ollama" on page 517, I showed many examples that didn't quite work. Pick any other AI model you like (open source or commercial, local or hosted) and try to get the following tasks to work:

Let's try ChatGPT, which is now well known.

26.1 Count the number of `r` letters in the word `strawberry`.

> *Prompt*: How many instances of the letter "r" are in the word "strawberry"?
>
> *ChatGPT*: The word **"strawberry"** contains **two instances** of the letter "r".

Urk. Let's see if ChatGPT knows how to write code to get the right answer.

> *Prompt:* Write a Python program to count the number of instances of the letter "r" in the word "strawberry".
>
> *ChatGPT*: Certainly! Here's a simple Python program to count the number of instances of the letter `"r"` in the word `"strawberry"`:
>
> ```
> # Define the word
> word = "strawberry"
>
> # Count the number of instances of the letter "r"
> count = word.count('r')
>
> # Output the result
> print(f"The letter 'r' appears {count} times in the word 'strawberry'.")
> ```
>
> # Explanation:
>
> - We use the `count()` method of a string to count how many times the letter `"r"` appears in the word `"strawberry"`.
> - The `count()` method returns an integer representing the number of occurrences of the specified substring (`'r'` in this case).
>
> # Output:
>
> ```
> The letter 'r' appears 2 times in the word 'strawberry'.
> ```
>
> Let me know if you'd like further clarification or modifications!

Weird! ChatGPT generates correct Python code that returns 3, but says it returns 2. I resubmitted the original question to ChatGPT a few days later:

> *Prompt:* How many instances of the letter "r" are in the word "strawberry"?

This time, the model gave the correct answer:

> *ChatGPT:* The word **"strawberry"** contains **3** instances of the letter **"r"**.

I don't know what's going on. Now ChatGPT used the digit **3** instead of the text **two** for the letter count.

26.2 Define a limerick.

> *Prompt:* What is a limerick?
>
> *ChatGPT:* A limerick is a short, humorous poem consisting of five lines with a specific meter and rhyme scheme. Here's what characterizes a limerick:
>
> Structure: Five lines
>
> Rhyme scheme: AABBA
>
> The 1st, 2nd, and 5th lines rhyme with each other and are typically longer (often 8–9 syllables).
>
> The 3rd and 4th lines rhyme with each other and are shorter (usually 5–6 syllables).
>
> Tone: Light-hearted, playful, and often silly or nonsensical
>
> May include wordplay, puns, or even a mildly risqué joke
>
> Example:
>
> ```
> There once was a man from Peru,
> Who dreamt he was eating his shoe.
> He awoke with a fright,
> In the middle of night,
> To find that his dream had come true!
> ```
>
> Let me know if you'd like help writing your own!

Success!

26.3 Write a limerick. You pick the subject.

Prompt: Write a limerick about a cat on the moon.

ChatGPT: Sure! Here's a limerick about a cat on the moon:

```
A curious cat on the moon,
Played jazz with a space-travel tune.
It purred in low G,
With zero gravity,
Then danced with a cheese-eating loon!
```

Want a different style—silly, spooky, or romantic?

Another success!

26.4 Write a Python function called `utc_now_str()` to return the current date and time in UTC as a string.

ChatGPT: Sure! Here's a Python function that returns the current date and time in UTC as a string:

```
from datetime import datetime

def utc_now_str():
    # Get current UTC time and format it as a string
    return datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S')

# Example usage
print(utc_now_str())
```

This function uses the `datetime.utcnow()` method to get the current time in UTC and then formats it as a string in the `YYYY-MM-DD HH:MM:SS` format.

Let me know if you'd like any adjustments!

That's correct, although ChatGPT didn't know that `utcnow()` will go away in the future.

26.5 Write a Python generator comprehension to return the integers 1 through 5.

ChatGPT: You can write a Python generator comprehension like this:

```
gen = (x for x in range(1, 6))

# Example usage: to print the values
for num in gen:
    print(num)
```

This creates a generator that will yield the integers from 1 to 5 when iterated over.

Let me know if you need more details!

This response is also correct.

26.6 Duplicate Example 1-5.

The entire boxed section that follows was the response from ChatGPT.

Prompt: Write a Python program that accesses the Wayback Machine at the Internet Archive. Prompt the user for a domain and date, then query the Wayback Machine (using the requests library) for its archived version of that website closest to that date. If one was found, print its URL and display the returned URL in the default web browser. Else, print an error message.

ChatGPT: Here's a complete Python program that does what you describe:

```
import requests
import webbrowser

def main():
    print("Wayback Machine Archive Finder")
    domain = input("Enter a domain (e.g. example.com): ").strip()
    date = input("Enter a date (YYYYMMDD): ").strip()

    url = (f"http://archive.org/wayback/available?" +
        f"url={domain}&timestamp={date}")

    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status()
        data = response.json()
```

```
            if "archived_snapshots" in data and \
                "closest" in data["archived_snapshots"]:
                archived_url = data["archived_snapshots"]["closest"]["url"]
                print(f"Archived version found: {archived_url}")
                webbrowser.open(archived_url)
            else:
                print("No archived version found for that domain and date.")

        except requests.RequestException as e:
            print(f"Error accessing Wayback Machine API: {e}")

    if __name__ == "__main__":
        main()
```

How it works

Prompts the user for a domain and a date in YYYYMMDD format.

Queries the Wayback Machine Availability API (*http://archive.org/wayback/available*).

If an archived snapshot exists, it prints the URL and opens it in the default web browser.

If none is found, it prints an error message.

Errors (network, invalid response, etc.) are caught and displayed.

Would you like me to also add support for cases where the user enters just a year or year+month (e.g. 2020 or 202003) instead of the full YYYYMMDD?

That response was really good! By the way, it was also much shorter than the original version returned by ChatGPT a few months ago.

26.7 Duplicate Example 24-3.

The entire boxed section that follows was the response from ChatGPT.

Prompt: Query the Internet Archive API for the movie titled "Eegah." Display the first ten matches, choose one at random, and display it in the web browser.

ChatGPT: Here's a small Python script that does exactly that—queries the Internet Archive for items with the title "Eegah," lists the first ten matches, picks one at random, and opens its details page in your default web browser.

```
#!/usr/bin/env python3
import requests
import random
import webbrowser
import sys

SEARCH_URL = "https://archive.org/advancedsearch.php"

def main():
    # Query: title "Eegah" and moving-image media
    params = {
        "q": 'title:"Eegah" AND '+
            'mediatype:(movies OR television OR "moving image")',
        "fl[]": ["identifier", "title", "year", "date",
            "publicdate", "mediatype"],
        "rows": 10,               # fetch first 10
        "page": 1,
        "output": "json"
    }

    try:
        r = requests.get(SEARCH_URL, params=params, timeout=20)
        r.raise_for_status()
    except requests.RequestException as e:
        print(f"Error talking to Internet Archive: {e}")
        sys.exit(1)

    data = r.json()
    docs = data.get("response", {}).get("docs", [])
    if not docs:
        print('No matches found for title "Eegah".')
        sys.exit(0)

    # Display the first ten matches
    print("First matches:")
    for i, d in enumerate(docs, start=1):
        year = d.get("year") or \
            (d.get("date") or d.get("publicdate") or "")[:4]
        title = d.get("title", "(no title)")
        mediatype = d.get("mediatype", "")
        print(f"{i:2d}. {title} ({year}) [{mediatype}]  ->" +
            f"https://archive.org/details/{d.get('identifier')}")

    # Pick one at random among those shown
    choice = random.choice(docs)
    chosen_url = f"https://archive.org/details/{choice['identifier']}"
    print("\nRandomly chosen item:")
    print(f"{choice.get('title','(no title)')} -> {chosen_url}")

    # Open in the default browser
    try:
        webbrowser.open(chosen_url)
    except Exception as e:
        print(f"Could not open browser automatically: {e}")

if __name__ == "__main__":
    main()
```

How to run

1. Save as eegah_ia_search.py.
2. python3 eegah_ia_search.py

This will print the first ten matches (if available), then randomly open one in your browser.

Like the previous exercise, this response was also correct, and more compact than the response ChatGPT gave me a few months ago. Have we reached the singularity yet, or do we need to wait a few weeks?

27. Performance

27.1 Add the numbers from 1 to 100 in various ways and then time them (see Example A-26):

- Use a `for` loop.
- Use pairs of numbers to compute the sum. Hint: look up the story about Gauss in school.
- Use the direct formula from these pairs to get the sum in one shot.
- Does Python have a built-in function to sum these numbers?

Example A-26. times.py

```
import time

def use_for():
    s = 0
    for n in range(1, 101):
        s += n
    return s

def use_pairs():
    # 1+100, 2+99, ... 50+51
    s = 0
    for n in range(1, 51):
        s += n + (101 - n) # or just 101
    return s

def use_formula():
    s = 50 * 101
    return s
```

```
def use_sum():
    return sum(range(1, 101))

def timer(func):
    t1 = time.time()
    s = func()
    t2 = time.time()
    print(f"{func.__name__:20}", s, f"{t2-t1:>20}")

timer(use_for)
timer(use_pairs)
timer(use_formula)
timer(use_sum)
```

```
$ python times.py
use_for              5050 7.867813110351562e-06
use_pairs            5050 5.4836273193359375e-06
use_formula          5050 2.384185791015625e-07
use_sum              5050 1.6689300537109375e-06
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

Bill Lubanovic has been busy with Unix since 1977, GUIs since 1981, databases since 1990, and the web since 1993:

- 1973–1980 (Chronobiology Labs): Circadian rhythm research under Franz Halberg.
- 1982–1988 (Intran): Developed MetaForm on the first commercial graphic workstation.
- 1990–1995 (Northwest Airlines): Wrote a graphic yield management system; got the airline on the internet; wrote its first internet marketing test.
- 1994 (Tela): Cofounded an early ISP; presented internet seminars.
- 1995–1999 (WAM!NET): Developed web dashboards and the 3M Digital Media Repository.
- 1999–2005 (Mad Scheme): Cofounded a web development/hosting company.
- 2002 (O'Reilly): Wrote parts of *Building Secure Servers with Linux*.
- 2005 (O'Reilly): Wrote parts of *Linux Server Security*.
- 2007 (O'Reilly): Coauthored *Linux System Administration*.
- 2010–2013 (Keep): Designed and built Core Services between web frontend and database backends.
- 2014 (O'Reilly): Wrote the first edition of *Introducing Python*.
- 2015–2016 (Internet Archive): Worked on APIs and a Python version of the Wayback Machine.
- 2016–2018 (CrowdStrike): Managed Python-based services processing billions of daily security events.
- 2019 (O'Reilly): Wrote the second edition of *Introducing Python*.
- 2019–2023 (Flywheel): Developed medical imaging systems.
- 2023 (O'Reilly): Wrote *FastAPI*.

Bill lives in the Sangre de Sasquatch hills of Minnesota with his wonderful family.

Colophon

The animal on the cover of *Introducing Python* is a python—that is, a member of the *Python* genus, which consists of 10 recognized snake species. Pythons are nonvenomous, constricting snakes native to the tropics and subtropics of the Eastern Hemisphere.

Pythons vary in length, based on species and sex, from approximately 3 feet long (ball python) to reported cases of more than 20 feet long (reticulated python). They are recognizable by their flat triangular-shaped heads and long backward-curving teeth. Generally, they have a combination of brown, tan, and black skin, arranged in diamond or interlocking blotch patterns. Albino pythons are white and yellow and, while at a disadvantage in the wild, are popular in zoos and as pets.

Pythons are strong swimmers, but these snakes almost exclusively ambush prey on land or in trees, attacking with their fangs and then immediately wrapping themselves around the quarry to asphyxiate it. Unlike boas, another well-known group of constrictor snakes, pythons lay eggs rather than birthing live young. The female python will brood over the eggs until they hatch, shivering with her large muscular coils to keep the eggs warm.

The Burmese python has become an invasive species in Florida, competing with the American alligator for prey and significantly reducing the number of native birds and small mammals in the Everglades. Along with the python's increased popularity as a pet in the 1990s, Hurricane Andrew may have been responsible for allowing captive Burmese pythons into the wild by destroying a zoo and breeding facility in 1992.

Most species of python have a conservation status of Least Concern, but the Burmese (native population) and Borneo short-tailed python are currently listed as Vulnerable. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Jose Marzan Jr., based on a black-and-white engraving from *Johnson's Natural History*. 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.