O'REILLY®

Third Edition

Introducing Python

Modern Computing in Simple Packages

Early Release

RAW & UNEDITED

Bill Lubanovic

O'REILLY®

Third Edition

Introducing Python

Modern Computing in Simple Packages

Early Release
RAW & UNEDITED

Bill Lubanovic

Introducing Python

THIRD EDITION

Modern Computing in Simple Packages

Bill Lubanovic

O'REILLY®

Introducing Python

by Bill Lubanovic

Printed in the United States of America.

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

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

Acquisition Editor: Brian Guerin

Development Editor: Corbin Collins

Production Editor: Kristen Brown

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

January 2026: Third Edition

Revision History for the Early Release

- 2024-10-14: First Release
- 2024-12-11: Second Release

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

978-1-098-17440-8

[LSI]

Dedication

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

Brief Table of Contents (*Not Yet Final*)

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) and *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 actually found useful in twenty 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 them to you. Areas like databases and the web are still covered, but technology changes fast. A Python programmer might now be expected to know something about cloud computing, machine learning, or event streaming. You'll find something here on all of these.

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

When you’re learning something new, it’s hard to tell which terms are specific rather than colloquial, and which concepts are actually 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.

NOTE

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

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

Audience

Although it helps if you’ve done some programming before, I want to make it possible for beginning programmers to get something out of 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.

- Featured FastAPI for web examples.
- Updated many examples and arcane tidbits[1].
- Expanded old Chapter 19 (*Be a Pythonista*) into a full Part 2.

Outline

Part 1 (Chapters 1–13) explains 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.

- *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 some 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)*: 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 some data comparison.
- *Chapter 7 (For and While)*: 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 (Sets and Dictionaries)*: *Dictionaries* (aka *dicts*) and *sets* let you save data by their values rather than their position. This turns out to be very 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.
- *Chapter 13 (Review)*: This is a look back at the preceding twelve chapters, tying together some concepts

Part 2 (Chapters 14-18) 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 14 (Development Environment)*: Here, we get into *virtual environments* (Python version control) and *package management* with `pip` and other tools.
- *Chapter 15 (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 16 (Testing)*: How do you know that your code works? Some techniques can save you a lot of time and grief.
- *Chapter 17 (Debugging)*: Sometimes you need to dig to find the problem.
- *Chapter 18 (Review)*: This refreshes information from the previous Tools chapters.

Part 3 (Chapters 19-30) shows how Python is used in specific application areas such as the web, databases, networks, and so on; read these chapters in any order you like.

- *Chapter 19 (Text Data)*: Go beyond the basic string description in Chapter 4: Unicode characters, *regular expressions* for text pattern matching, and more.

- *Chapter 20 (Binary Data)*: This area doesn't seem to get much discussion in Python books. You can do some tricky things in Python with non-textual data that you might think require a lower-level language like C.
- *Chapter 21 (Dates and Times)*: Dates and times can be messy to handle. This chapter shows common problems and useful solutions.
- *Chapter 22 (Files)*: Basic data storage uses *files* and *directories*. This chapter shows you how to create and use them.
- *Chapter 23 (Data in Time: Concurrency)*: This is the first hard-core system chapter. Its theme is data in time — how to use *programs*, *processes*, and *threads* to do more things at a time (*concurrency*). Python's *asyncio* is heavily featured.
- *Chapter 24 (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 25 (Data in a Box: Databases)*: 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 get larger and more complex, they need the services of *databases* — traditional *relational* ones, and some newer *NoSQL* data stores.
- *Chapter 26 (The Web)*: The web gets its own chapter — clients, servers, APIs, and frameworks. You'll *crawl* and *scrape* websites, and then build real websites with *request* parameters and *templates*.
- *Chapter 27 (Data Science)*: Python is right at home here, with a very wide variety of tools and methods that are used in production systems every day.

- *Chapter 28 (AI)*: This new chapter is devoted to the very 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 29 (Performance)*: This is another new chapter, showing various methods of speeding up Python in cases where it isn't frisky enough. It includes an introduction to a very recent Python superset called Mojo, which I think could have quite an impact if its development plans go well.
- *Chapter 30 (Review)*: As before, this chapter refreshes some info from the previous ones.

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. Oh, wait, it was. In 1977, I first encountered UNIX and C, and 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 early graphic user interface (GUI) systems in the early 80s, before the Mac and Windows. I worked with the early non-commercial Internet until it went public in the early 90s. After the release then of Mosaic (the first cross-platform browser), I figured out that the Web would be the "next big thing", and pivoted to web development.

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

One advantage of my early exposure to computing is that many of the things 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 thing 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 sometimes 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. Sometimes I do this to keep a code example simple and short, and the variable names are clearly throwaways. We don't want code lines to use too much space in the print edition. Paper doesn't grow on trees, you know[2].

And thanks for jumping in. One of my cats will 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.

`Constant width italic`

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

NOTE

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

WARNING

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 2026 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

environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

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

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.xhtml

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-2e*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

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

Watch us on YouTube: *http://www.youtube.com/oreillymedia*.

Acknowledgments

My sincere thanks to the reviewers and readers who helped make this better:

Corbin Collins,

---

1 A good name for a band, or pet food?

2 Oh, wait. It does.

Chapter 1. Introduction

Only ugly languages become popular. Python is the one exception.

—Don Knuth

You're here. Good. Then let's begin.

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

So, here we go.

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.
```

It 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 one in Figure 1-1.

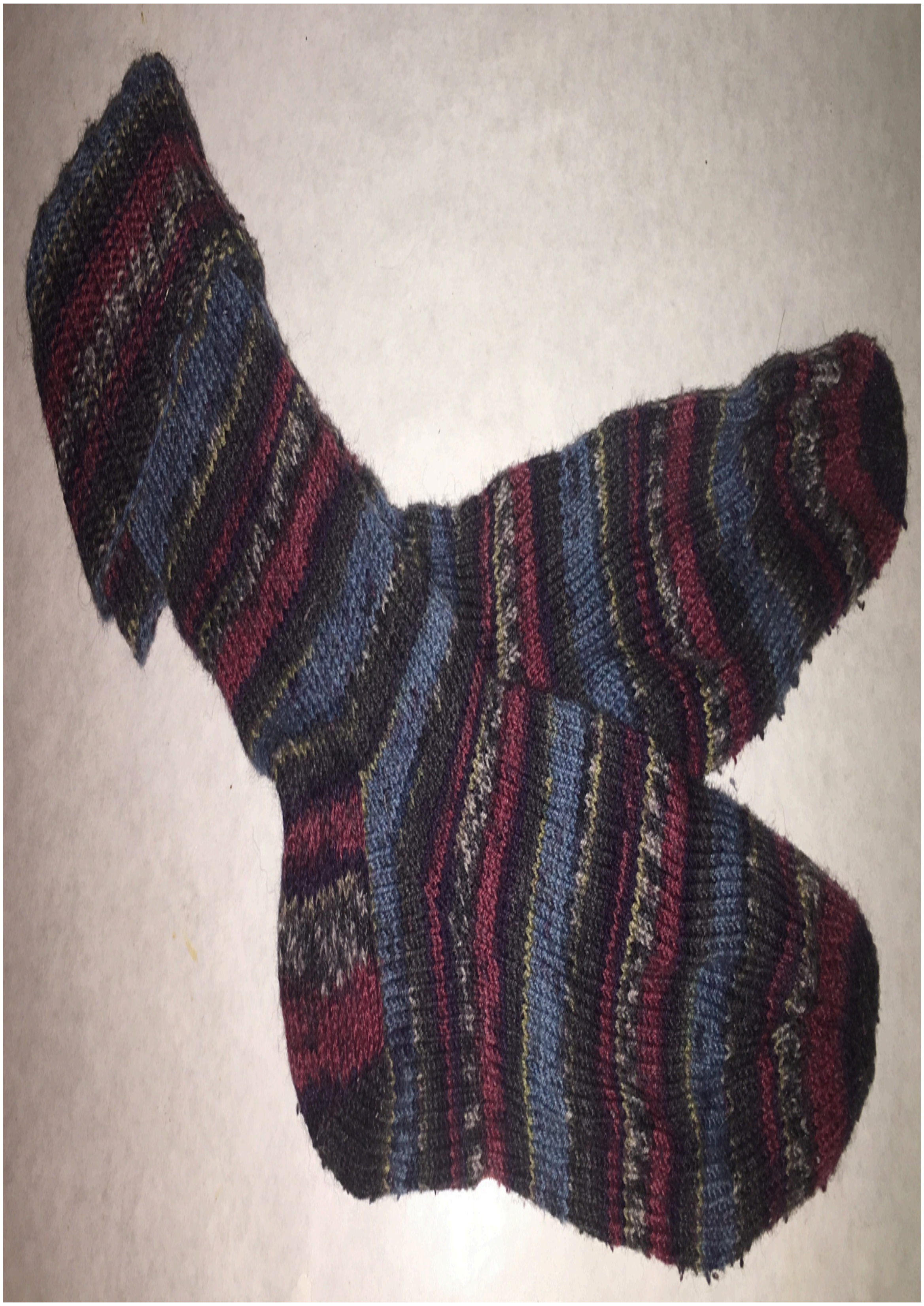


Figure 1-1. Knitted socks

This makes as much sense to me as a Sudoku puzzle does to one of my cats, 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 recognized 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.

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:

- A computer programming language
- Created in 1991 by the Dutch computer scientist Guido van Rossum
- Grown and guided by Guido 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 some 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 that we can do things with it. In this case, the program would print the fourth spell:

```
Expecto Patronum!
```

Why did we say `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`.

NOTE

People count from 1, so it might seem weird to count from 0. 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 very 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!
```

`quotes` is a variable that 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 things 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 brackets (`{` 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 *graphic user interfaces* (GUIs) for most operations, but to write programs, you'll need two things on your computer:

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

Chapter 14 will discuss fancier tools like *integrated development environments* (IDEs) that provide these features and more in one application, but in Part 1 we only need a text-based terminal window and an editor. These are all widely available on thr three main platforms that I mentioned above.

Coming right up, and 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 it, 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 the *major* version, *y* is a minor 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 ten years to replace the previous major version `2`. The minor versions come out roughly annually[2].

In your terminal window, type `python -V`. You should see something like this:

```
$ python -V
Python 3.13.0b3
```

> **NOTE**
>
> If your Python version is earlier than 3.13, watch in this book for notes that indicate things that may act differently in earlier versions.

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.

The Python Interactive Interpreter

Sometimes called the Python *shell*, this is what you get if you just type `python` (I've substituted *[version info]* below, 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* for you to type a line of valid Python code.

You type Python code after that prompt, and it executes it, line by line. This is a quick way to test snippets of Python, and I'll use it through 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?
>>>
```

`print()` is a built-in Python function 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. 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*:

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

Then, in your terminal window, tell Python to run it:

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

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

Built-In Python Features

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

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 Three. You access these from your program with the `import` statement, which is fully explained in Chapter 12, and you can see in in the example code Example 1-4 coming up below.

Third-Party Python Packages

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

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 1-4 below accesses the Wayback Machine and illustrates many of the things 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.)

Example 1-4. archive.py

```
1 import webbrowser
2 import json
3 from urllib.request import urlopen
4
5 print("Let's find an old website.")
6 site = input("Type a website URL: ")
7 era = input("Type a year, month, and day, like 20150613: ") + "0000"
8 url = f"http://archive.org/wayback/available?url={site}&timestamp={era}"
```

```
9 response = urlopen(url)
10 contents = response.read()
11 text = contents.decode("utf-8")
12 data = json.loads(text)
13 try:
14     old_site = data["archived_snapshots"]["closest"]["url"]
15     print("Found this copy:", old_site)
16     print("It should appear in your browser now.")
17     webbrowser.open(old_site)
18 except:
19     print("Sorry, no luck finding", site)
```

This little Python program did a lot in a few fairly readable lines. 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`.

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 to the variable `contents`.
11. *Decode* `contents` to a text string in JSON format, and assign to the variable `text`.
12. Convert `text` to `data`—Python data structures.
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 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 it 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.

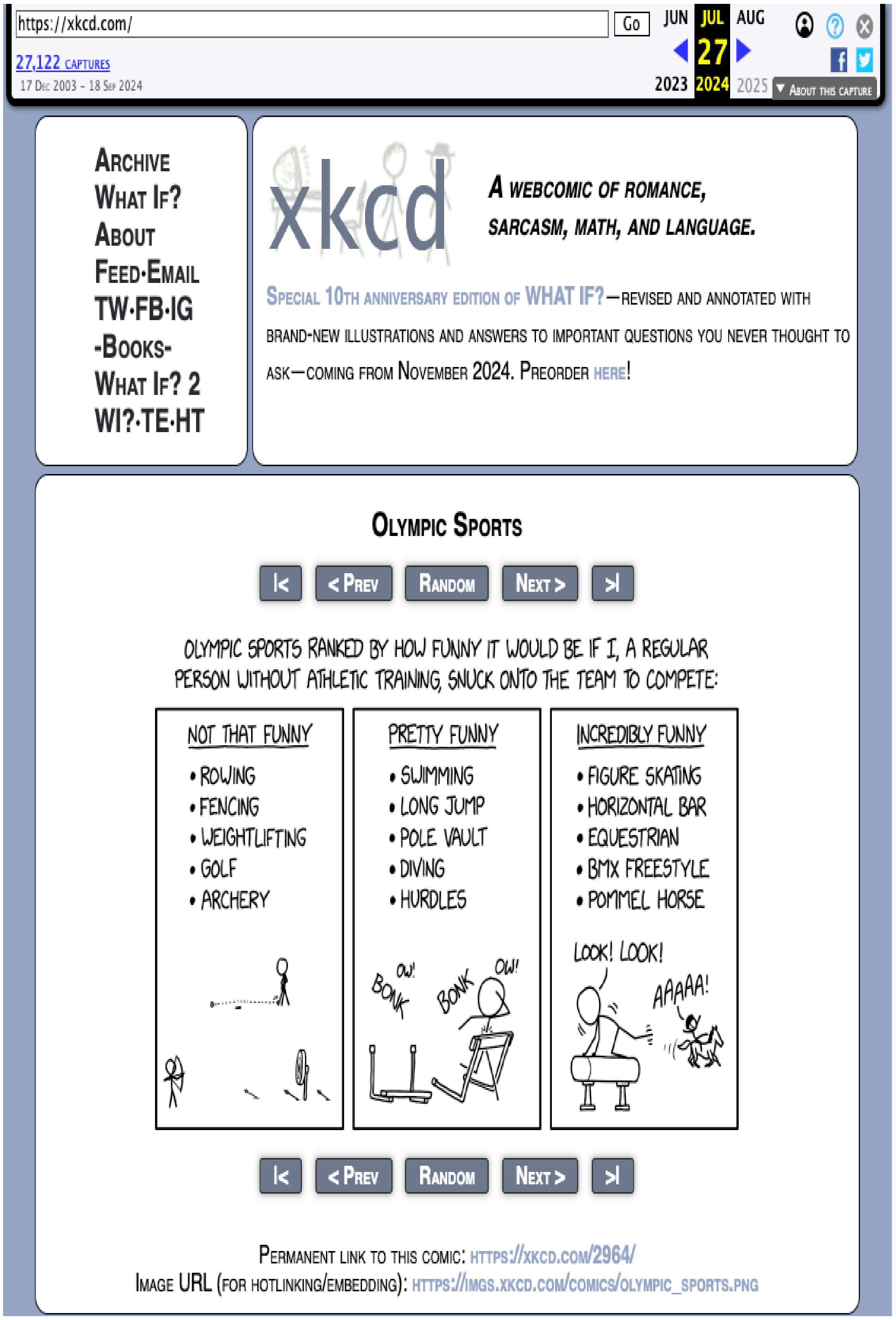
NOTE

Most computing languages use some 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 news site URL and a historic 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.



https://xkcd.com/ Go JUN JUL AUG 27 2023 2024 2025

27,122 captures

17 Dec 2003 – 18 Sep 2024

About this capture

Archive
What If?
About
Feed·Email
TW·FB·IG
-Books-
What If? 2
WI?·TE·HT

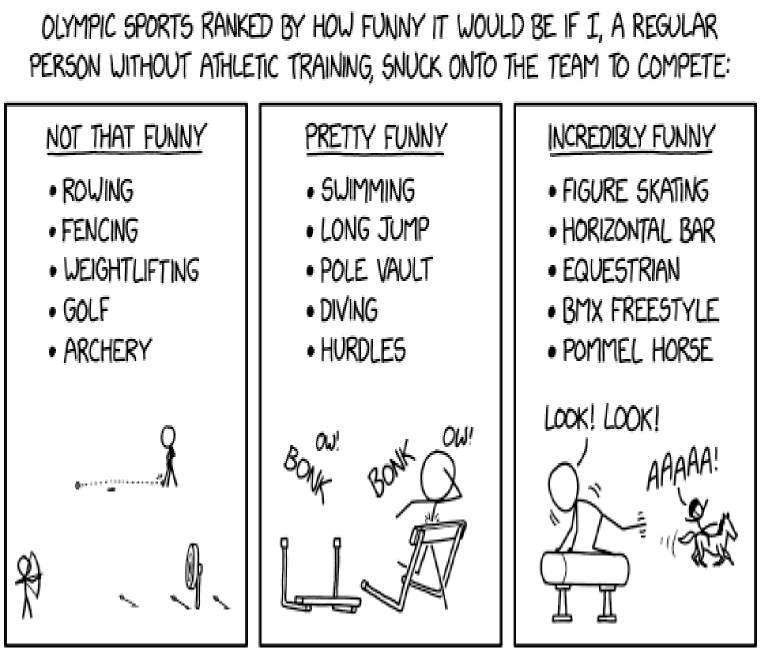
xkcd

A webcomic of romance, sarcasm, math, and language.

Special 10th anniversary edition of WHAT IF?—revised and annotated with brand-new illustrations and answers to important questions you never thought to ask—coming from November 2024. Preorder here!

Olympic Sports

|< < Prev Random Next > >|



OLYMPIC SPORTS RANKED BY HOW FUNNY IT WOULD BE IF I, A REGULAR PERSON WITHOUT ATHLETIC TRAINING, SNUCK ONTO THE TEAM TO COMPETE:

NOT THAT FUNNY	PRETTY FUNNY	INCREDIBLY FUNNY
• ROWING	• SWIMMING	• FIGURE SKATING
• FENCING	• LONG JUMP	• HORIZONTAL BAR
• WEIGHTLIFTING	• POLE VAULT	• EQUESTRIAN
• GOLF	• DIVING	• BMX FREESTYLE
• ARCHERY	• HURDLES	• POMMEL HORSE

|< < Prev Random Next > >|

Permanent link to this comic: https://xkcd.com/2964/
Image URL (for hotlinking/embedding): https://imgs.xkcd.com/comics/olympic_sports.png

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 the archived page above 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, and included a sample program and its output.

Ths next chapter gets into *variables* and data *types*, core concepts in any programming language.

---

[1] Usually only found in cookbooks and cozy mysteries.

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

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

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 Caveman Computer

Figure 2-1 is a drastically simplified diagram of a typical computer’s components.

Figure 2-1. Computer components

- The *central processing unit* (CPU, or “chip”) is the brains of the operation. Most computers now have multiple CPUs, and we’ll say more about this in Chapters 23 and 29.
- The *random access memory* (RAM) is the fast but limited storage area.
- A *disk* stores much more data than the RAM does, but is thousands of times slower.
- Common local *input* devices include keyboards, touchscreens, mice, cameras, and microphones.
- Common local *output* devices include displays and printers.
- A *network* communicates with other computers.

The CPU and RAM are *volatile*: they only work while electricity is flowing through them. The disk is *nonvolatile* (normally magnetic), and retains its data even when the whole computer is turned off. There’s a continual struggle between RAM (fast but limited) and disk (slow but much 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 something 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 29, a more complete computer architecture is discussed, including things like *caches* that affect performance.

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:

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

Figure 2-2 and Figure 2-3 are representations of the two bit possibilities.

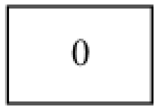


Figure 2-2. Unset bit

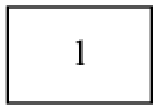


Figure 2-3. Set bit

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) different possible combinations of bit values in a byte.

Individual bits or bytes have no actual meaning. Each one has some combination of states, and we somehow need to keep track of them and treat them in a certain way. For example, because a byte can have up to 256 distinct states, we can use base 2, or *binary arithmetic*, to treat a byte as a tiny number. If all of its bits are 0 (off), that could represent the integer 0. If all bits are 1 (on), that could represent the integer 255. The least significant bit represents how many 1's, the next bit up represents how many 2's, and the top, most significant bit represents how many 128's. (This is like our familiar base 10 (*decimal*) system, where we have digits from 0 to 9, but no single digit for ten. Instead, we have a 1 in ten's *place*, and up to 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.

The bits in a byte have a particular *order*, from *least significant* to *most significant* (at the top). Figure 2-4 shows a byte with the integer value 0, Figure 2-5 represents the integer 1, and Figure 2-6 has all bits set, which could represent the integer 255.

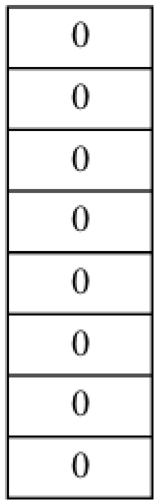


Figure 2-4. Byte with value 0

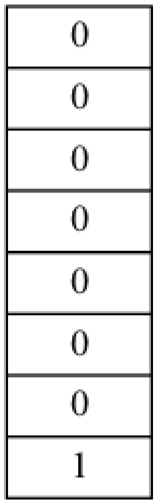


Figure 2-5. Byte with value 1

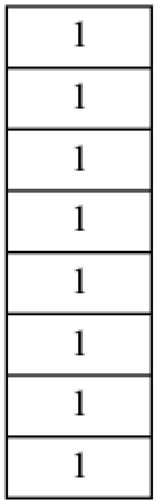


Figure 2-6. Byte with value 255

That example was an *unsigned* (always positive) number, 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 only have seven bits left to represent the number's *magnitude*, and can't get to 255 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 only need 62 distinct values to fit in a byte. The rest of the 255 possible bit combinations could be used for something else.

The ASCII standard was defined many decades ago to fill in these slots. It also included common punctuation symbols, and some non-printing 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 was what the A in ASCII stood for. In Chapter 4, you'll see how *Unicode* allows all of the world's languages (and symbols and even emojis) to be expressed in computers. There are millions of unique Unicode symbols — we can't stuff them all into a 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 2-7, four bytes in a row make a single four-byte number, with the value 1:

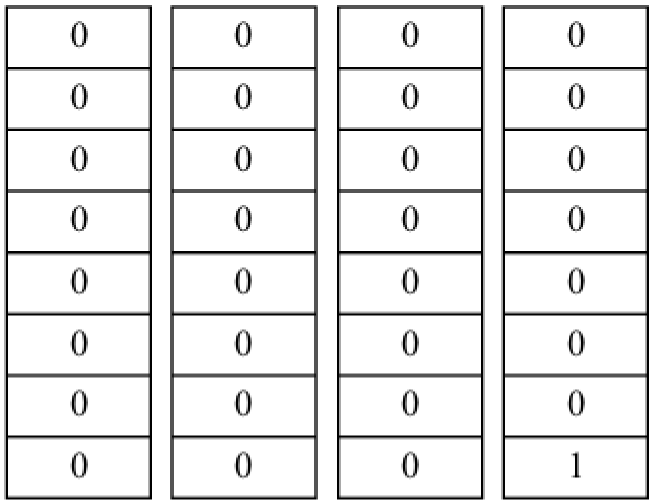


Figure 2-7. Four bytes, with value 1

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

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 only need 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.

Variables

This giant bookshelf of bytes in computer memory contains everything that the computer's brain (its *CPU*, or Central Processing Unit) needs to do computery things. But how do you keep track of everthing?

A Python program has code (the actual 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.

Python handles variables in a different way 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:

```
variable = value
```

For example:

```
x = 5
```

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

Try it in the Python shell:

```
$ python
Python 3.13.0b3 [version info]
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 5
>>> x
5
```

When you just type 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.

NOTE

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

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.

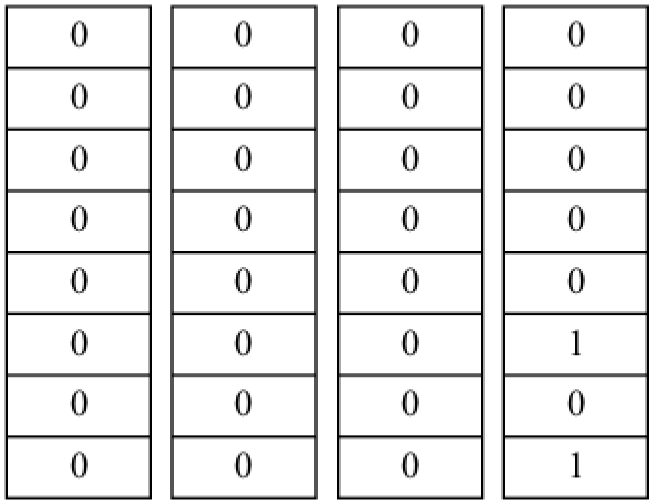


Figure 2-8. Four bytes, with value 5

Okay, 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 *most* programming languages (but *not* Python), a variable just 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 "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 the Python value for an integer with the value `5` actually looks like in memory.

NOTE

Many Python discussions mention that every data type in Python is implemented as an *object*. The problem is that the term "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 saying what's stored at each location. The actual number of bytes that each component takes up is an implementation detail that we don't need to care about:

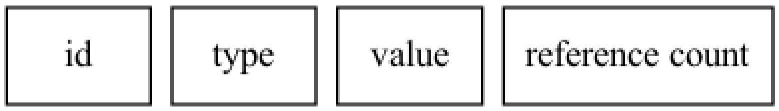


Figure 2-9. A Python object in memory

- The `id` is unique for each value object in memory. Although the Python intepreter may just use its physical location in memory, don't assume this. The important point is that it's 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 are bigger than single integers and need room to roam.

- What about that `reference count`? It's how Python tracks how many variables are pointing to the same value object. If this count reaches zero, then 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*.

Okay. 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`. It 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`.

Change the Value of a Variable

The value itself can be changed by assigning a new value to the same variable. So, if we now say:

```
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:

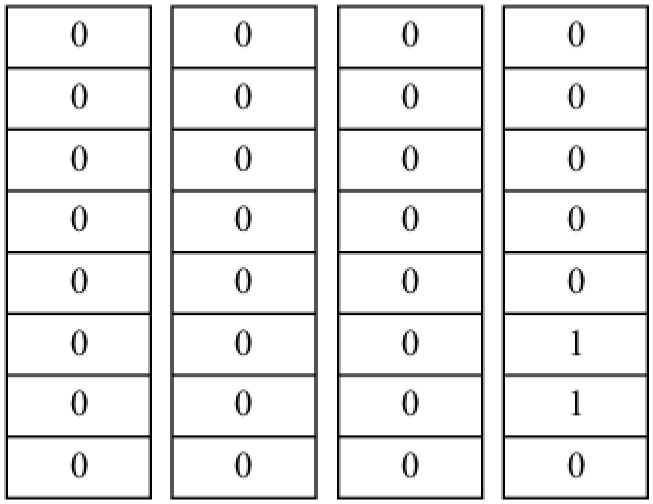


Figure 2-10. Four bytes, now with value 6

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

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

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

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

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

Delete a Variable

Another way to get rid of a value it 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 tbis 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.

If you've used other languages, this means you don't need to manage memory manually (say this fast three times) in Python. There are no C-style `malloc()` or `free()` calls. This is one of the *big* advantages of a language like Python; memory management is difficult and notoriously error-prone.

I went into 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 some rules.

- They can contain only these characters:
    - 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:

```
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`

These names, however, are not valid:

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

Naming Conventions

Python also has some *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.

- First, 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`.
- Even 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.
- Finally, although Python doesn't have true *constants* (variables that 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).
- Finally, 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 meemory 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 types to match these common hardware types. <Table 2-1 lists these built-in types, and which upcoming chapter describes them.

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

Table 2-1. Python's basic data types

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

Name	Type	Mutable?	Examples	Cha
			`o', 'drummer': 'Ringo'}`	

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* (like `5`) or a *variable* (like `x`, which was assigned the value `5`). There are rules for how to specify 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 addition of numbers).

Review/Preview

This chapter focused on *variables* (names used in programs) and *values* (the data that the names refer to). Python is different from many programming languages, because the variable is just a name, and the value includes other information, including its type, its actual data bits, and a count of how many variables refer to it.

NOTE

You can think of an *object* as a clear plastic box on that memory shelf, with some fixed-size contents (id, type, reference count), and some varying in size (the value bits themselves). A *variable* can be pictured as a sticky note attached to that object, or a tag on a thread taped 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)`**. It should echo `42` on the next line.

2.3 If you're still in the interactive interpreter, type **`43`**. It 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 what error messages 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* (whole numbers such as `42` 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 seeing the results immediately. 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 these 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 chapter in Chapter 10, but for now you just need to know that a function has:

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

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 Chapter 6 and Chapter 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 bases in “Bases”.

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 between `1` and `9`:

```
>>> 05
  File "<stdin>", line 1
    05
     ^
SyntaxError: invalid token
```

NOTE

This Python *exception* warns that you typed something that breaks Python’s rules. I explain what this means in “Bases”. 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`. See “Bases”.

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

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

To specify a negative integer, insert a – 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 just 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 normal 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
```

It 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 "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 7 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by z
```

Integers and Variables

All of the preceding examples used 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 = is calculated first, and then assigned to the variable on the left side.

If it helps, think of it this way:

- Subtract `3` from `a`

- Assign the result of that subtraction to a temporary variable
- 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. It's 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 it's between two numbers, it produces the remainder when the first number is divided by the second:

```
>>> 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 things 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 `**`, 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 just 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 it 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.

We generally 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." Unlike Roman numerals, Arabic numbers don't have a single character that represents "10" 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 how many 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*. `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
```

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

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

You can go 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`. `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, is 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 likely use 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 the `int()` function.

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

The preceding text string started with valid digit characters (`99`), but it kept on going with others that the `int()` function just wouldn’t stand for.

`int()` 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 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 zeroes, named in 1920 by a nine-year-old boy):

```
>>>
>>> googol = 10**100
>>> googol
100000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000
>>> googol * googol
100000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
```

A *googolplex* is `10**googol` (a thousand zeroes, if you want to try it yourself). This was a suggested name for Google before they decided on *googol*, but didn't check its spelling before registering the domain name `google.com`.

In many languages, trying this would cause something called *integer overflow*, where the number would need more space than the computer allowed for it, with various bad effects. 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 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
>>> 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
```

Floats are not exact

Because computers are *binary* (base two) inside instead of decimal (base ten), many numbers that we commonly use can't be represented exactly as floating-point values:

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

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

This means that 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)
```

Read fractions at python.org for details.

Decimals

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.

Another common trick is to multiply currency values to get more exact arithmetic. For example, represent dollar or euro amounts as pennies (multiply by 100), calculate with integer pennies as much as possible, and convert back to dollars 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 27, 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/Prview

This chapter showed how to represent Booleans and integer and floating point numbers, as well as some of the things 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 Appendix A.

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*. In this case, they're 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 white space or directives like linefeeds. 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 how to make and format simple text strings, using ASCII (basic character set) examples. Two important text topics are deferred to Chapter 19: *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*).

One last note before we dive in: strings are the first example in this book of a *sequence* in Python. A sequence is an ordered collection of elements. Some of the things that you can get from any sequence are:

- If 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).

Create with Quotes

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

```
>>> '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 \" 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 19, but the main thing to know for now is that it's *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 ½" × 3 ½".'
'A "two by four" is actually 1 ½" × 3 ½".'
```

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

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'
```

`print()` strips quotes from strings and prints their contents. It's meant for human output. It helpfully adds a space between each of the things 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 19 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 \".

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

Finally, there is the *empty string*, which 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 the `str()` function 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) just 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 include spaces explicitly. 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 with [ ]

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 some 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'
```

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 there are 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 a whole chapter 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`*`)`. There's a longer discussion of functions in Chapter 10.

You can use the built-in string `split()` function to break a string into a *list* of smaller strings based on some *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 was called `tasks` and the string function was called `split()`, with the single separator argument `','`. If you don't specify a separator, `split()` uses any sequence of white space 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. It 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 say `'\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 how many instances of the old substring to replace. It returns the changed string but does not modify the original string. If you omit this final count argument, it 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 19.

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'
```

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

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

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

Strip with strip()

It's very 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. `strip()` 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
```

But 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 of the characters in the poem either letters or numbers?

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

Nope, there were some punctuation characters.

Case

In this section, we look at some 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'
```

NOTE

Because strings are immutable, none of these examples actually changes the `setup` string. Each example just 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...'
```

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.

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 to produce reports, forms, and other outputs where appearances need to be just so.

Besides the functions in the previous 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 very simplest sequence is a `%` followed by a letter indicating the data type to be formatted.

Table 4-1. Conversion types

`%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, an integer:

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

A float:

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

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 such as `actor` goes right after that final `%`. Multiple 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 converted 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`) saying how many characters to print from the data value. If the conversion type is `f`, this specifies *precision* (how many 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` just 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 in "Old style: %". The simplest usage is demonstrated here:

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

The arguments to the `format()` function need to be in the 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` referred to the first argument, `place`, and `1` referred 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 printed 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 (default `' '`) to pad the value string if it's shorter than *minwidth*.
- An optional *alignment* character. This time, left alignment is the default. `'<'` also means left, `'>'` means right, and `'^'` means

center.

- An optional *sign* for numbers. Nothing means only prepend a minus sign (`'-'`) for negative numbers. `' '` 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*.

```
>>> 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:

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

It’s 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 brackets (`{}`) 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 brackets:

```
>>> 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 `':'`.

```
>>> 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 actually 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 19), but you can find all the details at the standard documentation link.

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 with others to make a new string. Say you wanted 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 Print each list question with its correctly matching answer, in the 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."
...     ]
```

4.3 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.
```

4.4 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}
```

4.5 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()`.

4.6 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.

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

4.8 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", 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`*`xx`* for nonprintable bytes 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):

```
>>> 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 chaacter 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 has a `fromhex()` convenience function, 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:

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

Notice that we needed to say `bytes.fromhex(hexstring)` instead of something like `hexstring.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 19, 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

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 with []

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 was th 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 19.)

Get a Slice

As with strings, 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 1 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 took the `bytes` values that `b1` and `b2` referred to, combined them into a brand new `bytes` value, then assigned 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 it 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 that you care to in Chapters 19 and 20.

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

Get One Byte

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, access multiple elements with a slice:

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

Modify One Byte by its Index

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 actually 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 left-hand side of the = assignment:

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

Insert a Byte with insert()

Use `insert(__index__, __value__)` to insert byte *value* at index *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` wire:

```
>>> 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!!!')
```

Review/Preview

Unlike text strings, which only have 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 …

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 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 *white space* 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, it 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
```

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

```
>>> 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 a little trick — if you're in the middle of paired parentheses (or square or curly brackets), Python doesn't squawk about line endings:

```
>>> 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.

NOTE

If you’ve programmed in other languages, note that you don’t need parentheses for the `if` test. For example, don’t say something such as `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 use tabs, or mix tabs and spaces; it messes up the indent count.

We did a number of things 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.`

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
```

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

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, in Table 6-2, these are all considered `False`:

Table 6-2. False equivalents

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. So, if you type:

```
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 vowelness 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:

```
>>> tweet_limit = 280
>>> tweet_string = "Blah" * 50
>>> diff = tweet_limit - len(tweet_string)
>>> if diff >= 0:
...     print("A fitting tweet")
... else:
...     print("Went over by", abs(diff))
...
A fitting tweet
```

With our new tuskiness (aka assignment expressions) we can combine these into one step:

```
>>> tweet_limit = 280
>>> tweet_string = "Blah" * 50
>>> if diff := tweet_limit - len(tweet_string) >= 0:
...     print("A fitting tweet")
... else:
...     print("Went over by", abs(diff))
...
A fitting tweet
```

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 really don't need to use `match`! It's sometimes a bit more compact — and, by some benchmarks, a bit faster.

This is actually called *structural* pattern matching, to distinguish it from pure text matching. You'll see the difference in the coming examples.

NOTE

This `match` is not the same as the `match()` function in the *regular expression* library that you'll see in Chapter 19.

Simple Matches

The simplest use is like the `switch` statement in languages like C. Except — you don't to 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 peak 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
```

Notes:

- The first pattern (`x, y`) contained the *guard* `if y > 6`. And that isn't true, so …
- The second pattern (`[2, 5]`) didn't match the second value (`6`) 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`, amd `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 between 1 and 10 and assign it to the variable `secret`. Then, select another number between 1 and 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`, `'too high'` if greater than `secret`, and `'just right'` if equal to `secret`.

6.2 Assign `True` or `False` to the variables `small` and `green`. Write some `if`/`else` statements to print which of these matches those choices: cherry, pea, watermelon, pumpkin.

---

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

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

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 assigned the value `1` to `count`. The `while` loop compared the value of `count` to `5` and continued if `count` was less than or equal to `5`. Inside the loop, we printed the value of `count` and then *incremented* 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 ended 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 breaking as soon as it’s found. The `else` would be run if the `while` loop completed but the object was 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
```

NOTE

This use of `else` might seem nonintuitive. Consider it a *break checker*.

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, more Pythonic way:

```
>>> 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` 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.
```

NOTE

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 some variable to indicate whether you found what you wanted in the `for` loop.

Generate Number Sequences with range()

The `range()` function returns 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()` similar to how to 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`.

Like `zip()`, `range()` returns an *iterable* object, 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 thing out of the way first. You may hear two different 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. :)

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')
```

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 for cases in which 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*.

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 things:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
```

Combine with +

This is similar to combining strings:

```
>>> ('Groucho',) + ('Chico', 'Harpo')
('Groucho', 'Chico', 'Harpo')
```

Duplicate with *

This is like repeated use of +:

```
>>> ('yada',) * 3
('yada', 'yada', 'yada')
```

Compare

This 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 things 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.

NOTE

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 earlier in “Split with split()”, use `split()` to chop a string into a list by some 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 indexes count backward from the end:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

NOTE

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 one. 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, the trick to reverse a list:

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

None of these slices changed the `marxes` list itself, because we didn't assign them to `marxes`. To reverse a list in place, use *`list`*`.reverse()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.reverse()
>>> marxes
['Harpo', 'Chico', 'Groucho']
```

NOTE

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 alright 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 gave us a new list of Marxes called `others`, and we'd like 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, you 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 how many 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 deleted `'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']
```

NOTE

`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 were the last reference to it.

Delete an Item by Value with remove()

If you're not sure or don't care where the item is 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()`. If you call `pop()` with an offset, it will return the item at that offset; with no argument, it 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']
```

NOTE

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 *LIFO* (last in, first out) queue. This is more commonly known as a *stack*. `pop(0)` would create a *FIFO* (first in, first out) 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
priorities!']
>>> 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 using `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
```

NOTE

If you check for the existence of some value in a list often 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 how many 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()" 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. `join()` is a string method, not a list method. 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*.
- 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']
```

`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 it to descending:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort(reverse=True)
>>> numbers
[4.0, 3, 2, 1]
```

Get Length with len()

`len()` 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? `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]
```

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, tuples, 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]]
```

`deepcopy()` 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 of 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 it's much more common to iterate over lists:

```
>>> 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` completed 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 was empty in this example, `for cheese in cheeses` never completed a single loop and its `break` statement was never executed.

Iterate Multiple Sequences with zip()

There’s one more nice iteration trick: 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
```

`zip()` stops when the shortest sequence is done. One of the lists (`desserts`) was longer than the others, so no one gets any pudding unless we extend the other lists. Read the next section (“Iterate Multiple Sequences with zip_longest()”) for a new function that gives you more control if the sequqnces 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, if you use plain `zip()`, it runs out of gas when the shortest sequence ends. There's a variant that keeps going until the longest one ends. `zip_longest()` 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 of 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 between `1` and `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.

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 what the second item is:

```
>>> 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 indexes:

```
>>> 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 less functions — there is no `append()`, `insert()`, and so on — because they 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") 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 it would appear to work because there's no exception if you type this:

```
>>> number_thing = (number for number in range(1, 6))
```

The thing between 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". 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 some 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 less 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 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 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.

7.10 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"
```

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.

Chapter 9. Sets and Dictionaries

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, anbd are also often vary 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 actually be any of Python’s immutable types: Boolean, integer, float, tuple, string, and others that you’ll see in later chapters. 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.

> **NOTE**
>
> 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 brackets (`{}`) around comma-separated *`key`* `:` *`value`* pairs. The simplest dictionary is an empty one, 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'}
```

NOTE

In Python, it’s okay 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 preceding example, when you’re typing keys and values within the curly braces. It just helps readability.

Create with dict()

Some people don’t like typing so many curly brackets and quotes. You can also create a dictionary by passing named arguments and values to the `dict()` function.

The traditional way:

```
>>> acme_customer = {'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
>>> acme_customer
{'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
```

Using `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'}
```

The section “Iterate Multiple Sequences with zip()” introduces you to the `zip()` function, which makes it easy to create these two-item sequences.

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 was 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 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. Here's an attempt by an anonymous programmer to add him, but he's botched 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 replaced 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 assigned the value `'Gilliam'` to the key `'Terry'` and then replaced it with the value `'Jones'`.

Get an Item by [*key*] or with get()

This is the most common use of a dictionary. You specify the dictionary and key to get the corresponding value: Using `some_pythons` from the previous section:

```
>>> 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 defined or added originally. 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
```

NOTE

In Python 2, `keys()` just returns a list. Python 3 returns `dict_keys()`, which is an iterable view of the keys which works with `for ... in`. This 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 actually *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

There’s more than one way to merge dictionaries, contrary to the 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 Python 3.5, there's a new way to merge dictionaries, using the ** unicorn glitter, 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'}
```

Actually, 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 the discussion of `deepcopy()` ("Copy Everything with deepcopy()") 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:

```
>>> 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 was 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 combines `get()` and `del`. You give `pop()` a key; if it exists in the dictionary, it returns the matching value and deletes the key-value pair. If it doesn't exist, it 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, if you make a change to a dictionary, it will be reflected in all the names that refer to it:

```
>>> 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 it by one name changes it for the others.

Copy with copy()

To actually copy keys and values from a dictionary to another dictionary and avoid this, 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, and 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 was 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 copied the values as-is, meaning `signal_copy` got the same list value for `'red'` that `signals` had.

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
```

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 how many 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:

```
>>> 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`.

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 only want to know that something exists, and nothing else about it. It's 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 enpty set with `set()`:

```
>>> empty = set()
>>> empty
set()
```

You can also create sets by surrounding an iterable data type with curly brackets (`{}`). But it 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[2].

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] )
>>> 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}
```

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}
```

Iterate with for and in

Like 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

This is the most common use of a set. 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.

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 take the union of two sets that have some keys in common. Because a set must contain only one of each item, 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`.

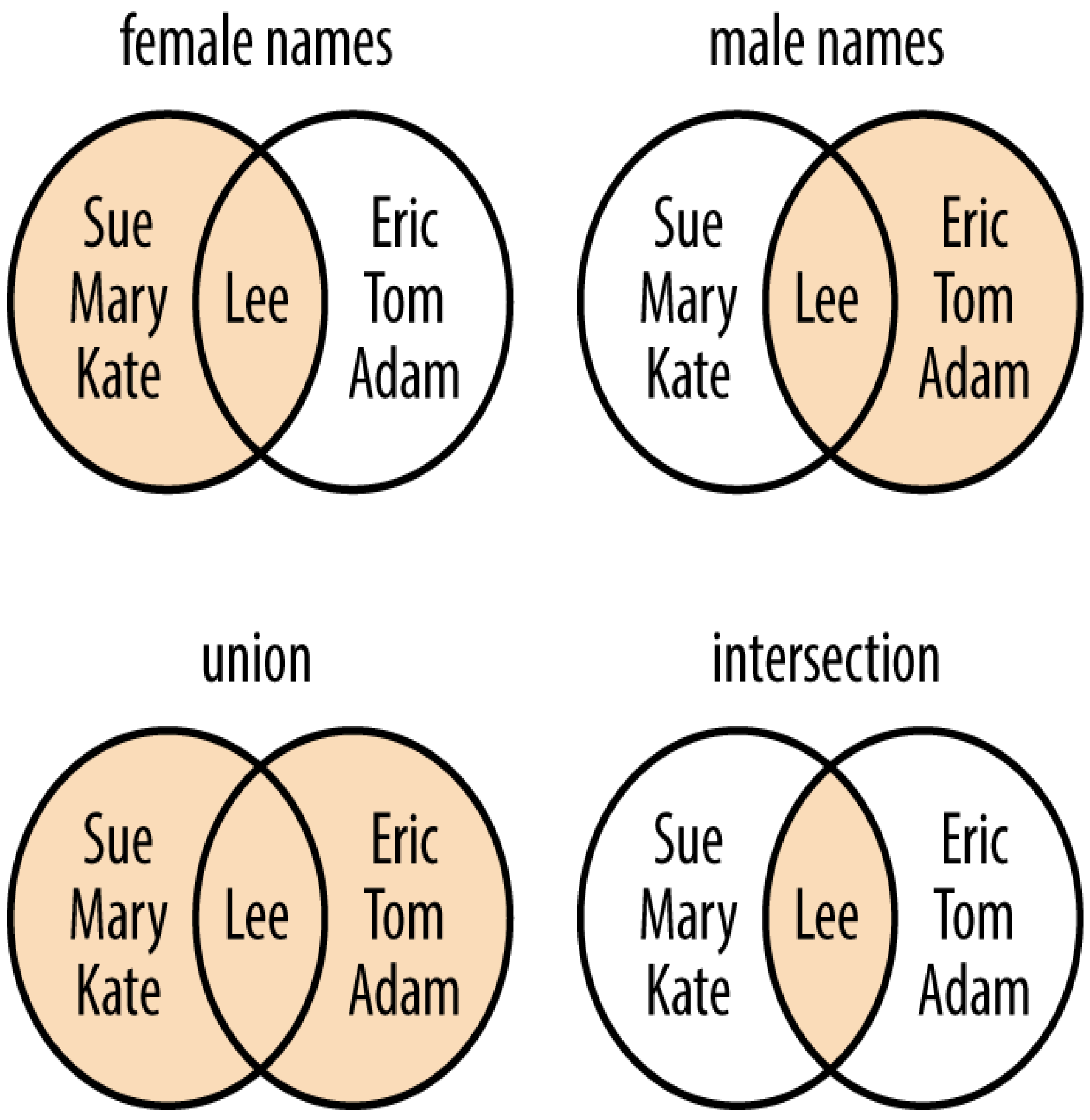


Figure 9-1. Common things to do with sets

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 of the items that appear in both lists that you compare. If neither of those ingredients were in `contents`, the `&` returns an empty set, which is considered `False`.

Now, let's rewrite the example from the previous section, in which we wanted 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 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[3].

```
>>> 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
```

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
```

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 particulzrly useful in data processing, when you want to group things by some 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']`

---

1 Also, the final score in the Strontium-Carbon game.

2 According to lawyers and exorcists.

3 Although, paraphrasing Groucho Marx, “I wouldn’t want to belong to a club that would have someone like me as a member.”

Chapter 10. Functions

The smaller the function, the greater the management.

—C. Northcote Parkinson

So far, all of 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 some way of organizing larger code into manageable pieces.

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 any number and type of output *results*.

You can do two things with a function:

- *Define* it, with zero or more parameters
- *Call* it, and get zero or more results

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 things 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
```

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 called the `make_a_sound()` function, Python ran the code inside its definition. In this case, it printed a single word and returned 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 made a big step. The combination of functions with tests such as `if` and loops such as `while` make it possible for you to do things that you could not do before.

Arguments and Parameters

At this point, it’s time to put something between those parentheses. Let’s define the function `echo()` 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.

NOTE

Saying it another way: they're called *arguments* outside of the function, but *parameters* inside.

In the previous example, the function `echo()` was called with the argument string `'Rumplestiltskin'`. This value was copied within `echo()` to the parameter `anything`, and then returned (in this case doubled, with a space) to the caller.

These function examples were pretty basic. Let's write a function that takes an input argument and actually does something 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 can return any number of output results (also including zero) of any type. If a function doesn't call `return` explicitly, the caller gets the result `None`.

```
>>> print(do_nothing())
None
```

None Is Useful

`None` is a special Python value that holds a place when there is nothing to say. It is not the same as the Boolean value `False`, although it looks false when evaluated as a Boolean. Here's an example:

```
>>> thing = None
>>> if thing:
...     print("It's some thing")
... else:
...     print("It's no thing")
...
It's no thing
```

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
```

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()`) are all `False`, but are not the same as `None`.

Let's write a quick function that prints whether its argument is `None`, `True`, or `False`:

```
>>> def whatis(thing):
...     if thing is None:
...         print(thing, "is None")
...     elif thing:
...         print(thing, "is True")
...     else:
...         print(thing, "is False")
...
```

Let's run some sanity tests:

```
>>> whatis(None)
None is None
>>> whatis(True)
True is True
>>> whatis(False)
False is False
```

How about some real values?

```
>>> whatis(0)
0 is False
>>> whatis(0.0)
0.0 is False
>>> whatis('')
 is False
>>> whatis("")
```

```
 is False
>>> whatis('''''')
 is False
>>> whatis(())
() is False
>>> whatis([])
[] is False
>>> whatis({})
{} is False
>>> whatis(set())
set() is False
>>> whatis(0.00001)
1e-05 is True
>>> whatis([0])
[0] is True
>>> whatis([''])
[''] is True
>>> whatis(' ')
  is 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 very common, a downside of positional arguments is that you need to remember the meaning of each position. If we forgot and called `menu()` with wine as the last argument instead of the first, the meal would 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.

Specify 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 actually be quite useful. Using 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'}
```

NOTE

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: it's 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.

Explode/Gather 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*. Nope, Python doesn’t have pointers.

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')
```

NOTE

When using `*`, you don’t need to name the tuple parameter `*args`, but it’s a standard idiom in Python.

Summarizing:

- 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
```

So:

- Outside the function, `*args` explodes the tuple `args` into comma-separated positional parameters.
- Inside the function, `*args` gathers all of the positional arguments into a single `args` tuple. You could use the names `*params` and `params`, but it's common practice to use `*args` for both the outside argument and inside parameter.

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.

Explode/Gather 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 function `print_kwargs()` to print its keyword arguments:

```
>>> def print_kwargs(**kwargs):
...     print('Keyword arguments:', kwargs)
...
```

Now try calling it 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.

Argument order is:

- Required positional arguments
- Optional positional arguments (`*args`)
- Optional keyword arguments (`**kwargs`)

As with `args`, you don't need to call this keyword argument `kwargs`, but it's common usage[1].

The `**` syntax is valid only in a function call or definition[2]:

```
>>> **kwparams
  File "<stdin>", line 1
    **kwparams
     ^
SyntaxError: invalid syntax
```

Summarizing:

- 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, `**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 Arguments

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 3 lets you specify *keyword-only arguments*. As the name says, they must be provided as *name=value*, not positionally as *value*. The single `*` in the function definition means that the following parameters `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)
a
b
c
d
e
f
>>> print_data(data, start=4)
e
f
>>> print_data(data, end=2)
a
b
```

Mutable and Immutable Arguments

Remember that if you assigned the same list to two variables, you could change it by using either one? And that you could not if the variables both referred to something like an integer or a string? That was because the list was mutable and the integer and string were immutable.

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[3]:

```
>>> 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.

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:

```
>>> 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

I’ve mentioned the Python mantra, *everything 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 things in Python that are difficult-to-impossible to carry out in many 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, it *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`

```
>>> 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 it 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 the function `run_something_with_args()`, the function name argument `add_args` was assigned to the parameter `func`, `5` to `arg1`, and `9` to `arg2`. This ended up running:

```
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. It's a built-in Python function that 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 it’s a single tuple:

```
>>> list_arg( (1, 2, 3) )
(1, 2, 3) <class 'tuple'>
```

As you saw earlier (“Explode/Gather Positional Arguments with *”) you can implode any number of arguments into a tuple paremeter 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 some complex task more than once within another function, to avoid loops or code duplication. For a string example, this inner function adds some text to its argument:

```
>>> def knights(saying):
...     def inner(quote):
```

```
...         return "We are the knights who say: '%s'" % 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')
```

Okay, 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 it, 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`

```
>>> 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 he (hypothetically) missed one of the stairs:

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

And for the function, this 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() + '!'
```

Mixing our ingredients:

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Finally, we get to the lambda. The `enliven()` function was so brief that we could 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 for cases in which you would otherwise need to define many tiny functions and remember what you called them all. In particular, you can use lambdas in graphical user interfaces to define *callback functions*.

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. If you recall, we already used one of them, `range()`, in earlier code examples to generate a series of integers. In Python 2, `range()` returns a list, which limits it to fit in memory. Python 2 also has the generator `xrange()`, 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*. It's a normal function, but it returns its value with a `yield` statement rather than `return`. Let's write our own version of `range()`:

```
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
```

It's a normal 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 generator object:

```
>>> for x in ranger:
...     print(x)
...
1
2
3
4
```

NOTE

A 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)
...
>>>
```

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 or curly brackets. It's 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 what 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 function `document_it()` 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 actually 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.

So, 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)
Start 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
```

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 get the value of the global variable *and* change it 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 change it, it changes a different variable also named `animal`, but this variable is 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 assigned 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. It 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()` contained only the local variable `animal`. The global namespace contained the separate global variable `animal` and a number of other things.

Uses of _ and __ in Names

Names that begin and end with two underscores (__) 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 do some things directly, and maybe call other functions. But what if a function calls itself?[4] 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?

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[5], no matter how deeply nested. 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]
```

Async Functions

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 a number of reasons, I've pushed the details on all things async to Chapter 23, which covers the overall issue of *concurrency*, because:

- Async is confusing.
- You don't actually need to ever write an async function, although you should understand how one works.
- There are alternative techniques, such as *threads* and *multiprocessing*.

Exceptions

In some languages, errors are indicated by special function return values. When things go south[6], 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.

It's good practice to add exception handling anywhere an exception might occur 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 your program down gracefully. If an exception occurs in some function and is not caught there, it *bubbles up* until it is caught by a matching handler in some calling function. If you don't provide your own exception handler, Python prints an error message and some information about where the error occurred and then terminates the program, as demonstrated in the following snippet:

```
>>> 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 things 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 there is an error, an exception is raised and the code inside the `except` block runs. If there are no errors, 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, it's best to provide a separate exception handler for each. 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 the 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. It 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 as err:
...         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 raised an `IndexError` as expected. Entering `two` annoyed the `int()` function, which we handled in our second, catchall `except` code.

Make Your Own Exceptions

The previous section discussed handling exceptions, but all of 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.

> **NOTE**
>
> This 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 didn't even define any behavior for `UppercaseException` (notice we just used `pass`), letting its parent class `Exception` figure out what to print when the exception was 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'`.

---

1 Although *Args* and *Kwargs* sound like the names of pirate parrots.

2 Or, as of Python 3.5, a dictionary merge of the form `{**a, **b}`, as you saw in Chapter 5.

3 Like the teens-in-peril movies where they learn “The call’s coming from inside the house!”

4 It’s like saying, “I wish I had a dollar for every time I wished I had a dollar.”

5 Another Python interview question. Collect the whole set!

6 Is this Northern Hemispherism? Do Aussies and Kiwis say that things go “north” when they mess up?

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 custom data structure containing both data (variables, called *attributes*) and code (functions, called *methods*). It represents a unique instance of some concrete thing. Think of objects as nouns and their methods as verbs. An object represents an individual thing, and its methods define how 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. `8` is a different object. This means there's an integer class 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, you can have multiple objects (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 it 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 actually do something useful.

> **NOTE**
>
> 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 say:

```
>>> class Cat:
...     pass
```

Just as with functions, we needed to say `pass` to indicate that this class was empty. This definition is the bare minimum to create an object.

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 assigned them to the names `a_cat` and `another_cat`. But our `Cat` class had no other code, so the objects that we created from it just sit there and can't do much else.

Well, they can do a little.

Attributes

An *attribute* is a variable inside a class or object. During and after an object or class is created, 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", 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` was 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. There are also *class attributes*, and you’ll see the differences later in “Class and Object Attributes”.

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” and “Method Types”.

Initialization

If you want to assign object attributes at creation time, you need the special Python object initialization method `__init__()`:

```
>>> class Cat:
...     def __init__(self):
...         pass
```

This is what you’ll see in real Python class definitions. I admit that the `__init__()` and `self` look strange. `__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`.

But even this second `Cat` class definition didn't create an object that really did 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 it to a function as an argument, or return it as a result.

What about the `name` value that we passed in? It was 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; it's 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*.

NOTE

You can make many individual objects from a single class. But remember that Python implements data as objects, so the class itself is an object. However, there's only one class object in your program. If you defined `class Cat` as we did here, it's like the Highlander—there can be only one.

Inheritance

When you're trying to solve some 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 some additions or changes. It's 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.

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
...
>>> 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 it 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` inherited the `exclaim()` method from `Car`. In fact, `Yugo` says that it *is* a `Car`, which might lead to an identity crisis. Let's see what we can do about that.

NOTE

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, it's often recommended to emphasize other techniques like aggregation and composition. We get to these alternatives in this chapter.

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! Much like a Car, but more Yugo-ish.")
...
```

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! Much like a Car, but more Yugo-ish.
```

In these examples, we overrode 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 was not present in its parent class. Going back to classes `Car` and `Yugo`, 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 it wanted 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() 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 used `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).

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 things, 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`) class

It’s 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 listed 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!'`.

Mixins

You may include an extra parent class in your class definition, but as a helper only. That is, 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. For an example, I'll show how you would call an object's method, and what Python actually does behind the scenes.

Remember class `Car` from earlier examples? Let's call its `exclaim()` method again:

```
>>> a_car = Car()
>>> a_car.exclaim()
I'm a Car!
```

Here's what Python actually 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 it 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, 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 what if someone misbehaves?

```
>>> 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`. We don't want people to access this directly, so we define two methods: a getter (`get_name()`) and a setter (`set_name()`). Each is accessed by a property called `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*.

There are two ways to do this. 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 method, you add some decorators and replace the 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 actually 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'
```

NOTE

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”, 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
```

There's one more advantage of using a property over direct attribute access: 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 of 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 everything 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 it completely private, but Python does *mangle* the attribute 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 it becomes:

```
>>> fowl._Duck__name
'Donald'
```

Notice that it didn'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 if you change the value of the attribute in the child object, it 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'
```

But it 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 of its objects. Within a class definition, a preceding `@classmethod` decorator indicates that that 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 how many object instances 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 referred to `A.count` (the class attribute) in `__init__()` rather than `self.count` (which would be an object instance attribute). In the `kids()` method, we used `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 in 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 didn'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()` just 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 didn't change how `QuestionQuote` or `ExclamationQuote` were initialized, so we didn'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 different 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*, after the 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?

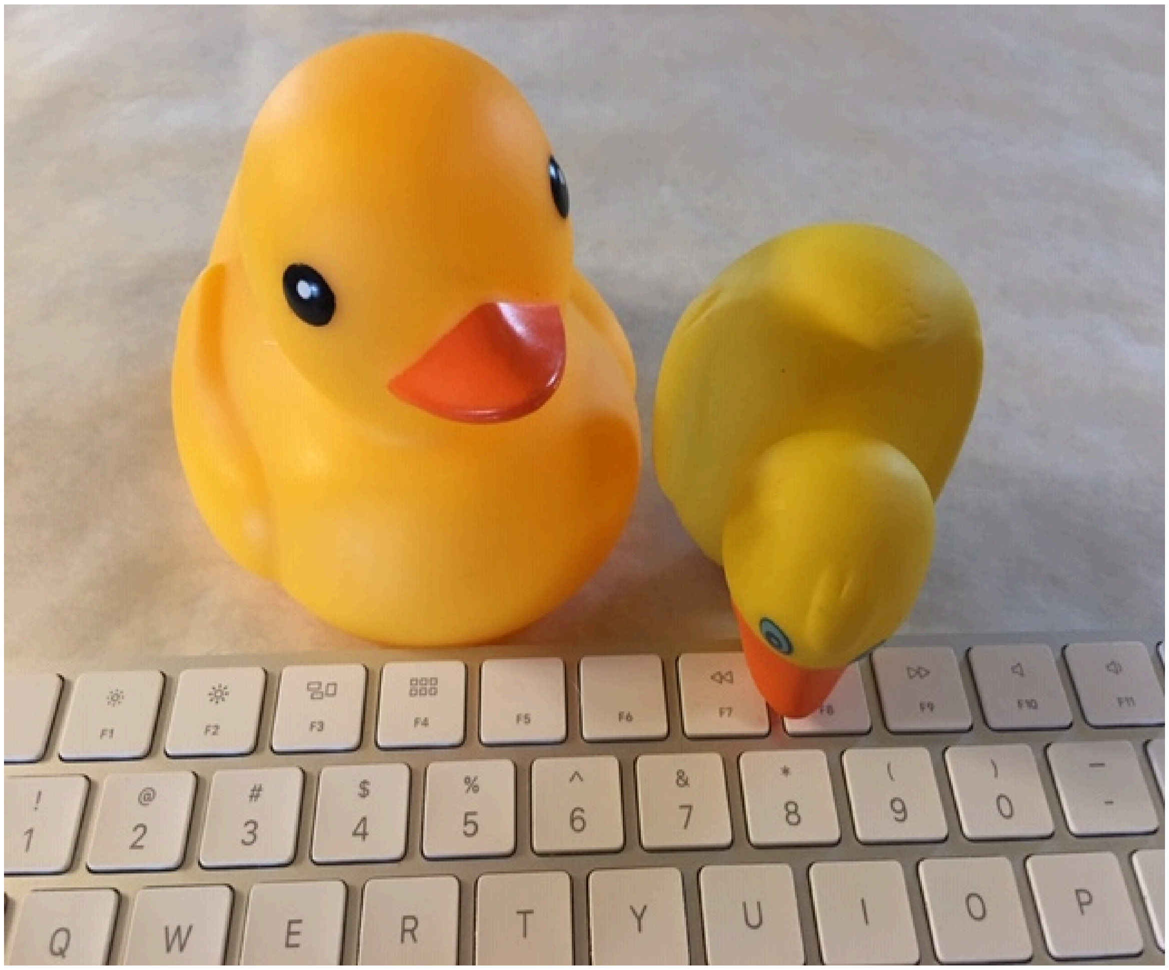


Figure 11-1. Duck typing is not hunt-and-peck

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 get at 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 very 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 to 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()`. `self.text` is the text string that this `Word` object contains, and the `equals()` method compares it 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 different 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 defined 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 it 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__()`. Table 11-1 and Table 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 19. 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 both `__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
```

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). It's tempting to build elaborate inheritance hierarchies, 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 a number of 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*.)
- If you have a number of variables that contain multiple values and can be passed as arguments to multiple functions, it might be better to define them as classes. 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.

  Guido's 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

- A newer alternative is the *dataclass*, in "Dataclasses".

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:

```
>>> 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 it 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 using *variable annotations* of the form `name: type` or `name: type = val`, like `color: str` or `color: str = "red"`. (Also see Chapter 15 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` defined a default value for its `teeth` attribute, so we didn’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 things for you automatically, so you don’t have to write them, and this can be a big time saver. See this guide 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—things 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 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 actually 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.

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`.

101.4 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` 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.

---

[1] Or even if you don’t want to.

[2] You’ll see many examples of double underscores in Python names; to save syllables, some people pronounce them as *dunder*.

[3] An inexpensive but not-so-good car from the ’80s.

[4] A mule has a father donkey and mother horse; a hinny has a father horse and mother donkey.

[5] Can you keep a secret? Apparently, I can’t.

[6] But see my earlier quote from Guido himself.

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 how 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.

The text of this book is organized in a hierarchy: words, sentences, paragraphs, and chapters. Otherwise, it would be 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.

Import a Module

The simplest use of the `import` statement is `import` *`module`*, where *`module`* is the name of another Python file, without the *.py* extension.

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 function called `choice()`:

The module 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 it.

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, it 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 used imports in two different places:

- The main program *lunch.py* imported our new module `fast`.
- The module file *fast.py* imported the `choice` function from Python's standard library module named `random`.

We also used imports in two different ways in our main program and our module:

- In the first case, we imported the entire `fast` module but needed 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 some other 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 imported 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)
```

Like 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 called `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 only wanted the `choice()` function from the `random` module.

Like the module itself, you can use an alias for each thing 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 sources import fast, advice

print("Let's go to", fast.pick())
print("Should we take out?", advice.give())
```

That `from sources` makes Python look for a directory named *sources*, starting under your current directory. Inside *sources* it 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

```python
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

```python
from random import choice

answers = ["Yes!", "No!", "Reply hazy", "Sorry, what?"]

def give():
    """Return random advice"""
    return choice(answers)
```

NOTE

If your version of Python is earlier than 3.3, you'll need one more thing in the *sources* subdirectory to make it a Python package: a file named *__init__.py*. This can be an empty file, but pre-3.3 Python needs it to treat the directory containing it as a package. (This is another common Python interview question.)

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. Actually, it looks in other places, as well, and you can control this.

Earlier, we imported the function `choice()` from the standard library's `random` module. That wasn't in your current directory, so Python needed 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 3.7 on my Mac:

```
>>> import sys
>>> for place in sys.path:
...     print(place)
...

/Library/Frameworks/Python.framework/Versions/3.7/lib/python37.zip
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload
```

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 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 will 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 it's in the directory above you: `from .. import rougarou`.
- If it's under a sibling directory called *creatures*: `from ..creatures import rougarou`.

The `.` and `..` notation was 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.

Namespace Packages

You’ve seen that you can package Python modules as:

- 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 things for 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 started 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 decided on US locations *north* and *south*, the files and directories would 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?

They 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`. `stuff` 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 only changed 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.

There’s only one copy of any module 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 things, 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 there’s a standard module that 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* are also very 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:

```
>>> periodic_table = {'Hydrogen': 1, 'Helium': 2}
>>> periodic_table
{'Hydrogen': 1, 'Helium': 2}
```

If the key was *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}
```

`defaultdict()` 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})
```

Okay, 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 chose the common element (`'eggs'`) with the lower count. This makes sense: breakfast offered 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'` was again common to both. Unlike addition, union didn't add their counts, but selected 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
```

NOTE

Starting with Python 3.7, dictionaries retain keys in the order in which they were added. `OrderedDict` is useful for earlier versions, which have an unpredictable order. The examples in this section are relevant only if you're a version of Python earlier than 3.7.

An `OrderedDict()` remembers the order of key addition and returns them 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

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. Here, we 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, it 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 used this as a simple illustration of deques. If you really wanted a quick palindrome checker, it would be a lot simpler to just compare a string with its reverse. 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

`itertools` 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.

`chain()` 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
```

`cycle()` 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.

`accumulate()` 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
```

`randrange()`, 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) between 0.0 and 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
- readthedocs

You can find many smaller code examples at activestate.

Almost all of 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 is a review of all of Part One: the basic data types and control structures of Python.

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']`.

---

1 At least, a little less readable than it already is.

2 Or am I? Bwa ha ha.

About the Author

Bill Lubanovic has been busy with Unix since 1977, GUIs since 1981, databases since 1990, and the web since 1993:

- 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; and wrote its first Internet marketing test.
- 1994 (Tela): Cofounded an early ISP.
- 1995–1999 (WAM!NET): Developed web dashboards and 3M Digital Media Repository.
- 1999–2005 (Mad Scheme): Cofounded a web development/hosting company.
- 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.
- 2020 (O’Reilly): Wrote the second edition of *Introducing Python*.
- 2019-2023 (Flywheel): ...
- 2023 (O’Reilly): Wrote FastAPI

Bill enjoys life in the Sangre de Sasquatch mountains of Minnesota with his wonderful family:

- Tom, Roxie, and Lettie
- Karin, Erik, and Maeve
- Critters: Tonks, Beans, Chester and Lucy.

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 birth 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 increased popularity as a pet in the 1990s, it is thought that Hurricane Andrew in 1992 is responsible for allowing captive Burmese pythons into the wild by destroying a zoo and breeding facility.

Most species of python have a conservation status of Least Concern, but the Bermese (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, based on a black and white engraving from *Johnson's Natural History*. 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.