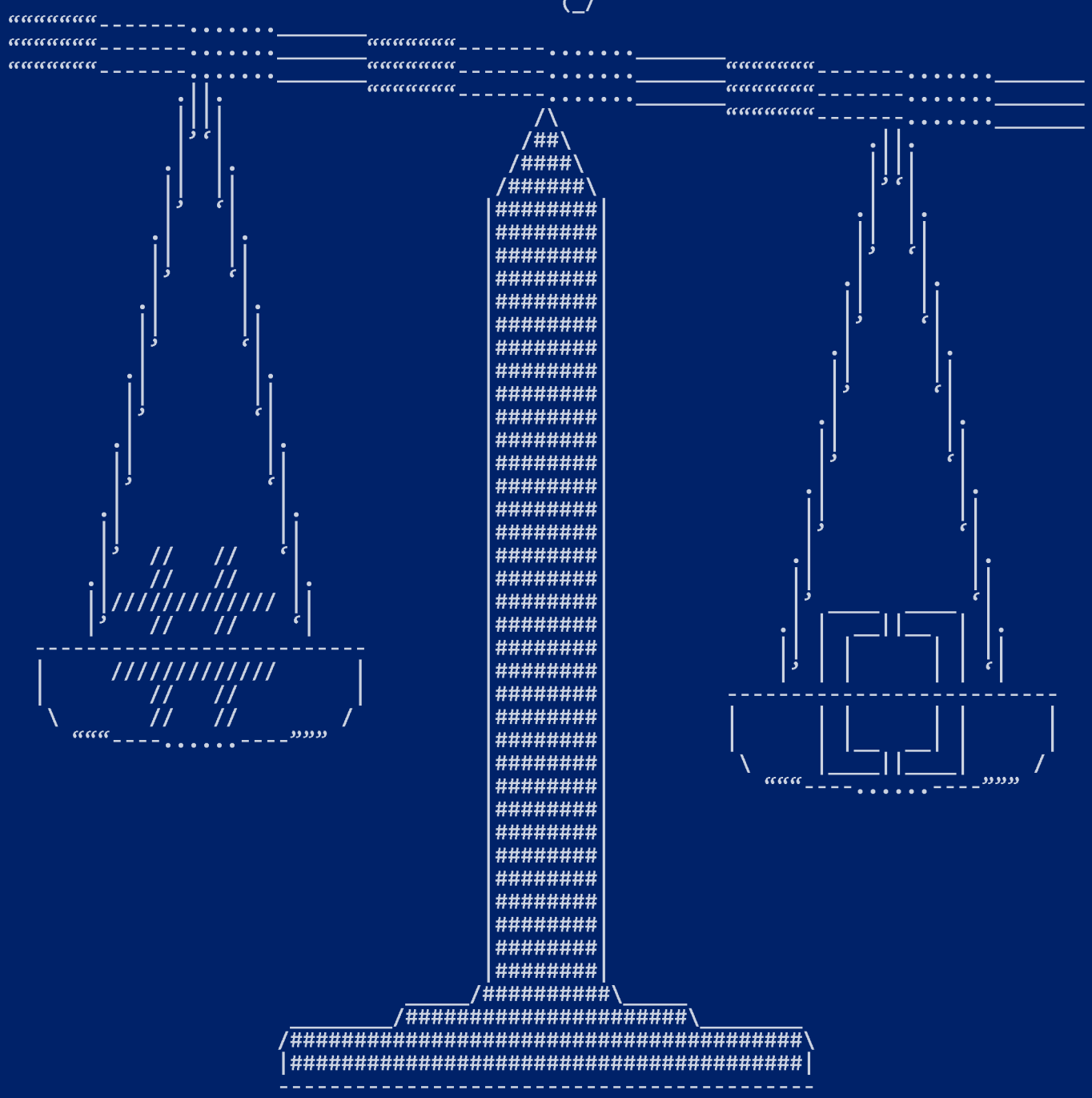
Computer Programming for Babies



Jonathan Fairfield

Paul Orn

Contents

Preface

No lawyers were harmed in the writing of this book.

Part I

Programming in Python

Chapter 1

What is Programming?

Computer programming might seem intimidating at first. It can conjure images of hackers in dark rooms staring at neon green text flying across black screens, or mathematicians writing equations filled with Greek letters on chalkboards. Rest assured that programming is far more straightforward (and far less like a Hollywood thriller) than these impressions might suggest. Programming involves simply writing a set of directions for the computer to follow. Anyone who has ever read a recipe is already well prepared to program.

In this chapter we'll introduce you to how programs work with two illustrative analogies: baking cookies and supplying precise instructions for guiding a directionally-challenged friend to his target destination.

1.1 As Easy As Baking

A computer program is, at its heart, like a recipe—a set of steps that, when followed carefully and supplied with the right ingredients, produce a specific result. Recipes are composed of a list of instructions, which you—the cook—read from top to bottom. Some instructions include simple tasks that must be completed exactly once: "place a bowl on the counter" or "add flour, sugar, eggs, butter, and chocolate chips." Others require repetition: "keep stirring until ingredients are well-mixed" or "scoop each teaspoonful of dough onto the cookie sheet." Some steps should be performed conditionally: "remove from the oven when golden brown." When put together in the proper order, they form a recipe that will lead to a batch of cookies.

```
1 Place a bowl on the counter.
2 Add flour, sugar, eggs, butter, and chocolate chips.
3 Until ingredients are well mixed:
4     Stir ingredients.
5 While there is still dough left:
6     Scoop a teaspoonful of dough.
7     Place scoop of dough on baking sheet.
```

8 Place baking sheet into oven.
9 Remove from the oven when golden brown.

Baking is extremely similar to programming. Unlike bakers, though, computers are exact; given the same ingredients and recipe, they will always produce identical results. Unfortunately, this also means that a computer will faithfully obey your instructions even when it is wrong. A computer cannot infer your intent and will happily compute the wrong result or do something nonsensical if it is so commanded. In the recipe above, for example, a computer working on line 7 will be unconcerned if the scoops of dough are placed in one big pile rather than spaced out so they can bake evenly. Worse, a computer could take line 9 to mean that it should wait until the oven is golden brown; in the meantime, your house is likely to burn down.

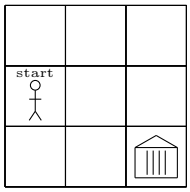
As you gain programming experience, you'll get better at detecting when the computer has interpreted your instructions differently from the way you intended. These *bugs*, as they are known, are a very common occurrence. Even expert programmers rarely write perfect code at the first attempt, and a large portion of programming time is actually spent *debugging*—refining your initial attempt into something correct.

Our cookie recipe offers one final lesson about programming: natural human language is far too imprecise to accurately instruct computers. In later chapters, we will explore vocabulary and syntax specially designed to unambiguously communicate programmer intent to computers—*programming languages*. In the meantime, however, we will first focus on the high-level computational thinking that you will employ every time you code.

1.2 A Simple Program for Giving Directions

Over the next few pages, we will give a sense for what a program looks like and how to build one. To do so, we will consider the plight of Larry, a hypothetical lawyer with a shockingly poor sense of direction. He appears in court often but struggles to find his way from his parking space to the courthouse. As his good friend, you have the honor of answering his phone calls when he gets lost and, thereby, of helping him find his way to court.

Larry's world takes the form of a grid (see below). Today, Larry needs your help getting from the far left of the grid to the courthouse in the lower right. You must give him directions.



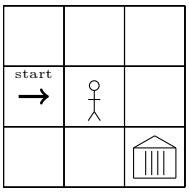
The only instructions you can give him are to move up, down, left, or right. We can think of Larry as our computer; he will follow any directions we give

him to the letter. Our directions will make up a program—a set of instructions to accomplish a task (getting Larry to a courthouse) based on a particular input (Larry's initial position). The *language* in which we write these programs has exactly four commands—the four directions we can tell Larry to move (up, down, left, and right).

To get Larry one step closer to the court house, we can start by telling him to move right. Our command for this task is as follows:

```
1 move right
```

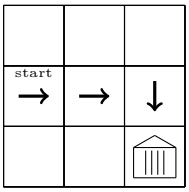
Now he's two steps away from the courthouse.



With the commands we have at our disposal, no single instruction will get Larry all the way to the courthouse from his parking spot. We'll have to give him more than one command.

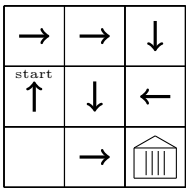
```
1 move right
2 move right
3 move down
```

Larry will follow these instructions line by line from top to bottom until he runs out, at which point he will stop wherever he happens to be. (This is exactly the same way we interpreted our cookie recipe earlier: start at the top and follow each instruction in sequence until reaching the bottom.) By moving right twice and then down once, Larry will successfully arrive at the courthouse.



This certainly isn't the only program that will get him to the proper destination. You might instead tell him to follow a more circuitous route, moving up, right twice, down, left, down again, and then right to arrive at the courthouse.

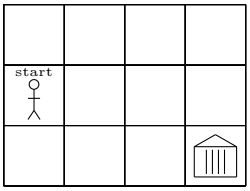
```
1 move up
2 move right
3 move right
4 move down
5 move left
6 move down
7 move right
```



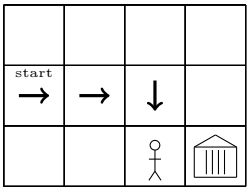
These instructions, while technically correct, are decidedly less direct than our first program. Larry will be pleased to have made it to the courthouse, but he will be frustrated that you wasted so much of his time in the process. The same is true with computer programs, where there are many ways to instruct a computer to solve any problem, but some methods are more efficient, easy to understand, or compact than others.

1.3 A Less Simple Program for Giving Directions

The next day, your ever directionally-challenged friend Larry finds himself lost after parking a little further away than yesterday. This time, he is four spaces from the courthouse.

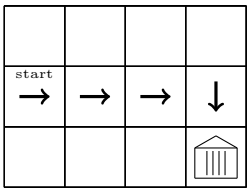


If he follows your instructions from yesterday, he will end up one space away from the courthouse:



Clearly, you will need to give him a new set of directions. You could base your instructions on the directions from yesterday, this time telling him to move right *three* times and down once.

1 move right
2 move right
3 move right
4 move down



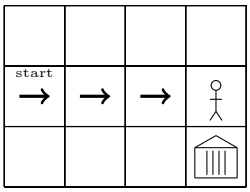
But what if he parks even further away tomorrow? Or what if he is lucky enough to park right in front of the courthouse? You will quickly get tired of giving directions that are so similar every day. Instead, it would be convenient to give Larry one set of instructions that will get him to the correct place every time he parks. In programming, we would call this *generalization*. We should aim to write programs that apply as widely as possible so that we don't have to continually revise them every time circumstances change slightly.

1.3.1 Repeating Instructions with Loops

Let's apply that principle to Larry, and start the solution by creating a program that gets him to the courthouse from any space in the *middle row*. Rather than spelling out each step, you can tell him to keep moving to the right until he is one space above the courthouse, at which point he should move down exactly once.

```
1 while not above the courthouse:
2       move right
3 move down
```

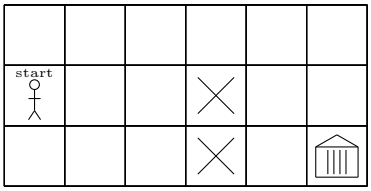
The first line of these instructions tells Larry to repeat whichever steps are indented immediately afterward so long as the condition "not above the courthouse" remains true. This indentation indicates which steps should be repeated. The subsequent instructions that are not indented are only to be followed after the repetition is done. In computer programs, we refer to this repetition as a *loop*. In this situation, he will move right, check that he is not above the courthouse, move right again, check again, and finally move right and check one last time.



At this point, he sees that he is above the courthouse, breaking the condition on line 1 and allowing him to proceed to the next direction after the loop, line 3. He will move down a single time and arrive at the courthouse.

1.3.2 Debugging

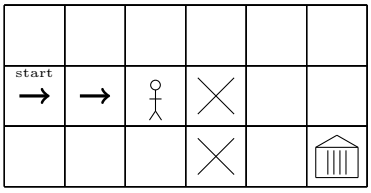
Unfortunately, you forgot that periodic construction in front of the courthouse can interfere with the instructions you have just given.



When Larry tries to follow your instructions, he will run into a problem at the construction site. Specifically, he will call you on the phone to say:

```
Error on line 2: Tried to move right but was unable due to a construction
    site in the way.
```

You find him here, to the immediate left of the construction site.



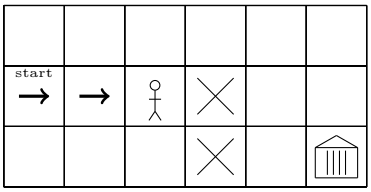
Retracing Larry's steps, he initially checked whether he was above the courthouse (line 1), found that he was not, and moved right (line 2). He returned to the beginning of the loop (line 1), checked again, found that he still wasn't above the courthouse, and moved right (line 2). Finally, he returned to line 1, again found that he wasn't above the courthouse, and tried to move right (line 2). However, his move right was impeded by the construction. Unable to follow your instruction, he simply gave up and stopped.

You need to fix the program so that it works in this new situation. These sorts of unintended misbehaviors and errors are known as *bugs*, and the act of altering a program to eliminate them is known as *debugging*. To debug your previous program will need to add instructions to avoid the construction site, making your program a little more complicated. First, Larry needs to move right until he reaches the construction site, then he'll need to go around it by moving up, to the right, and down. He can then continue moving right again just like before.

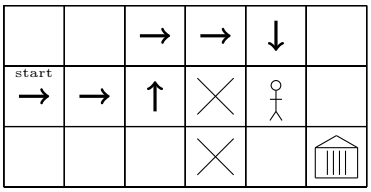
```
1 while not adjacent to construction:
2     move right
3 move up
4 move right
5 move right
6 move down
```

```
7  while not above the courthouse:
8       move right
9  move down
```

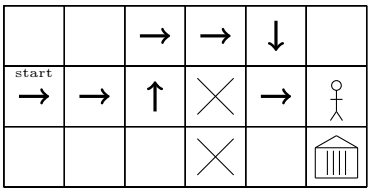
The first loop (on line 1) tells Larry to keep moving right until he is standing in a square adjacent to the construction site.



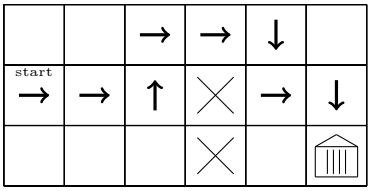
When he reaches the construction site, the condition "not adjacent to construction" will no longer be true. He will therefore exit the loop and proceed to line 3. Lines 3 through 6 navigate Larry around the construction site.



Finally, once Larry is on the other side, the instructions are the same as they were when there was no construction. Larry enters another loop (line 7) that instructs him to keep moving right so long as he is "not above the courthouse."



When he is above the courthouse, this condition no longer holds, so he exits the loop and continues to the final line, which tells him to move down and gets him to the courthouse.



Upon seeing these instructions, Larry will probably start to complain. There are a lot of steps, and it's difficult, at first glance, to understand why exactly each step is necessary. He has a point. If you need to go back and modify these instructions, it will probably take you a long time to figure out what you intended when you first wrote them. Anyone else who tries to read them will have an even harder time. When programming, presentation matters: code is a writing composition task, and future readers, yourself included, will be grateful if you make your programs as easy to understand as possible. This practice, called *style*, will be a recurring theme throughout this book.

1.3.3 Documenting Your Program

To meet this need, programmers use *comments*, which are lines of code that the computer should ignore. They are written purely for the purpose of *documenting* the logic behind your code for yourself and other readers. To help Larry make sense of our instructions, we'll add comments. A # sign at the beginning of a line tells your computer (in this case, Larry) that everything that comes afterward on that line is just for documentation and should not be executed. Comments make it possible to explain your code to readers in natural human language without altering your instructions to the computer. To make things easier to read, you can also add a few empty lines to the instructions to space things out better. Larry (and your computer) will just ignore these blank lines.

```
1  # Move right until you reach the construction site.
2  while not adjacent to construction:
3      move right
4
5  # Go around the construction site.
6  move up
7  move right
8  move right
9  move down
10
11 # Continue right.
12 while not above the courthouse:
13     move right
14
15 # Arrive at the courthouse.
16 move down
```

The program above provides the exact same directions as the program we wrote before. However, the version with the comments is dramatically easier for your reader to understand. More importantly, a few weeks later, when Larry inevitably runs into trouble again and you have long forgotten the layout of the courthouse property, you will have an easy time getting back up to speed with your instructions.

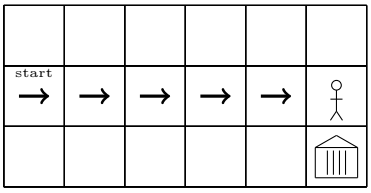
1.3.4 Conditional Behavior with If-Statements

Thinking you have the problem solved, you leave Larry with the updated instructions. A couple of weeks later, he calls you, completely exhausted, hours after his courthouse appointment. It turns out he is halfway across the city. When you inquire as to why, he angrily responds that he followed your directions exactly, moving to the right until he was next to construction. It turns out that the courthouse construction crew had taken the day off, so, finding no construction near the courthouse, Larry kept moving right until he stumbled upon a construction site on the other side of town. Your carefully crafted instructions were perfect so long as there was construction, but would fail when there wasn't.

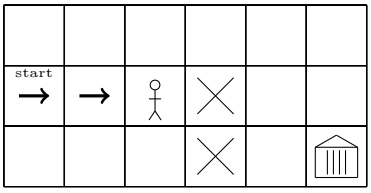
You need to give the hapless Larry instructions that will work no matter whether there is construction. First, you know that he needs to repeatedly move to the right. There are two conditions under which he should stop—if he is above the courthouse or if he is next to construction. We change our instructions to the following:

```
1 # First, move right as far as you can.
2 while not above courthouse and not next to construction:
3     move right
```

This loop instructs Larry to move right repeatedly, and to stop as soon as either (1) he is above courthouse:



or (2) is next to construction.



In other words, he should keep looping until one of these conditions is broken.

Reviewing your instructions, you realize they are slightly ambiguous. Larry could interpret your instruction to mean exactly what you intend:

```
1 # First, move right as far as you can.
2 while (not above courthouse) and (not next to construction):
3     move right
```

The parentheses indicate the logic of the instruction: Larry should keep moving right so long as he is both (1) not above the courthouse and (2) not next to construction. On the other hand, it might have been reasonable for him to think that you meant:

```
1  # First, move right as far as you can.
2  while not (above courthouse and not next to construction):
3      move right
```

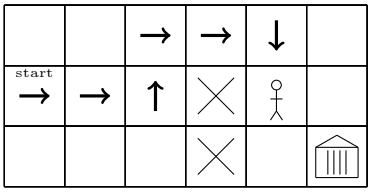
The logic of this interpretation takes a moment to parse. Larry will keep moving to the right so long as he is neither (1) above the courthouse nor (2) not next to construction. If we rearrange the double negatives in this instruction, we find that Larry will keep moving to the right so long as he is either (1) not above the courthouse or (2) next to construction. In other words, you just told Larry to wander into a construction site.

Since you clearly meant the first interpretation, we'll use the version with parentheses that clarify that intent. It's always good style to ensure that your meaning is entirely unambiguous. (Remember the example of the ambiguous cookie recipe!)

When Larry finally exits this loop, he could be in two possible situations. If he is next to construction, he needs to go around it and keep moving right; if there isn't any construction, he needs to stay exactly where he is. You can express this logic with a *conditional* command that will execute the indented instructions only if the condition holds.

```
5  # Go around the construction site if there is one.
6  if next to construction:
7      move up
8      move right
9      move right
10     move down
```

Unlike a loop, the indented code after a conditional executes only once. If the *condition* (in this case, "next to construction") is true, Larry will follow the indented directions in order exactly once and then continue to line 11.



If the condition is not true (Larry is not next to construction), he will simply skip all of the indented directions and go straight to line 11. Since conditionals begin with the word "if," they are often referred to as *if-statements*.

Finally, now that he has bypassed any construction, Larry can keep moving right until he is above the courthouse and then go down to finish his journey.

```
12 # Continue right.
13 while not above the courthouse:
14     move right
15
16 # Arrive at the courthouse.
17 move down
```

If he were already above the courthouse, the loop on line 13 will never execute, since the condition "not above the courthouse" does not hold; in that case, he would skip directly to line 17.

Reviewing our entire program:

```
1  # First, move right as far as you can.
2  while (not above courthouse) and (not next to construction):
3      move right
4
5  # Go around the construction site if there is one.
6  if next to construction:
7      move up
8      move right
9      move right
10     move down
11
12 # Continue right.
13 while not above the courthouse:
14     move right
15
16 # Arrive at the courthouse.
17 move down
```

1.4 Summary

Although you would probably be quite frustrated with Larry by the end of this ordeal, he did nothing wrong. He carefully followed every direction that you gave him, sometimes so faithfully that your seemingly ironclad instructions were too imprecise.

Just like your instructions for Larry, computer programs consist of lines of code that a computer follows in order from start to finish.

Programs can have loops to repeat operations until a particular exit condition is met. They can have conditionals to account for a wide range of possible inputs (the programmatic equivalent of construction sites) and change the behavior of the program accordingly. They also include comments to allow programmers to *document* their code. Python, specifically, requires each comment

to be preceded with a # character and distinguishes the code inside loops and conditionals from the rest of the program using indentation.

In two chapters, we will begin writing real Python programs, which you should find to be a familiar and comfortable task after your experience with Larry. First, however, you will need to install a few tools so that your computer can write, understand, and execute programs in Python.

Chapter 2

Installing and Using Python

To create and run Python programs, you'll need to install some software. All of this software is free and easy to install. Here, we'll provide instructions for Windows, macOS, and the popular Ubuntu variant of Linux.

Specifically, you will install:

- Python, the interpreter that gives your computer the ability to understand the Python language and run the code you write.
- Pip, a package manager that makes it possible to add new capabilities to Python, like reading PDF documents and interacting with websites.
- IDLE, an editor specifically designed for composing Python programs (much like how Notepad, Word, and PowerPoint are editors built for creating other kinds of documents). For Windows and macOS, IDLE will automatically download when you install Python 3.

Once you've installed everything you need, we'll guide you through the creating, editing, and running Python programs. By the end, you should be ready to dive into writing code.

2.1 Installing Python

Here you'll install Python for whatever operating system you use. Skip to the section relevant to you. Note that we'll install Python 3, which we'll be using throughout the book, as opposed to Python 2. The distinction between Python 2 and Python 3 is important because not all programs that work for one version will work in the other. In practice, both versions 2 and 3 are still popular, but Python 3 is the most up-to-date version. Python 3 is increasingly becoming the prevalent choice for new Python development going forward, which is why we'll use it here.

2.1.1 On Windows

In your web browser, go to *www.python.org/downloads* to access the download page of the official Python website. You will be presented with two options—to download the latest version of Python 3 or the latest version of Python 2. Click the button for Python 3. A file should automatically begin to download. Once it is finished downloading, double click the file to begin the installation. On the first screen of the installation window, be sure to check the box next to "Add Python 3.X to PATH." Finally, click on "Install now" to begin the installation.

2.1.2 On macOS

In your web browser, go to *www.python.org/downloads* to access the download page of the official Python website. You will be presented with two options—to download the latest version of Python 3 or the latest version of Python 2. Click the button for Python 3. A *.pkg* file will automatically begin to download. Once it is finished downloading, open the file to begin the installation. Follow the installer's prompts to complete the installation.

2.1.3 On Ubuntu

Open the terminal and enter each of the following commands, pressing Enter at the end of each one. If additional prompts appear asking you whether you are sure you wish to install any of these programs, confirm that you wish to do so.

$ *sudo apt-get install python3*
$ *sudo apt-get install idle3*
$ *sudo apt-get install python3-pip*

You can check that Python has installed correctly by running the following command at the terminal:

$ *python3 –version*

Your computer should respond by displaying the name of the version of Python you have installed:

$ *python3 –version*
Python 3.X.X

You may need to use the command `python` rather than `python3`.

2.2 The Shell

We'll typically interact with Python programs on the *shell*. The shell is a text-based interface for giving your computer instructions for many of the same tasks you would usually accomplish with a mouse—exploring files, running programs, and composing documents. The shell is known by a variety of different names, including the terminal, the command line, and the command prompt. Here

we'll look at which shell to use on each operating system and practice using it for several important tasks you'll perform throughout this book.

2.2.1 Opening the Shell

Skip to the instructions relevant to your OS here.

Windows

In Windows, there are actually two different shells: the command line and Powershell. We'll be using PowerShell. Open PowerShell by clicking the Start button and typing in *powershell*. This should show a desktop app called "Windows PowerShell" as a result; click on this result to open PowerShell. You should now see a blue window with a few lines of white text. At the end of the last line is a blinking cursor, which is where you can type in commands.

```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Users\Larry>
```

macOS

In macOS, the shell is known as the *terminal*. Open Finder, click **Go** in the menu bar, click **Utilities** in the dropdown menu, and then click the "Terminal" icon in the Finder window to open it. Alternatively, you can initiate a Spotlight search by typing *cmd + space*, beginning to type "Terminal," and pressing *enter* to launch Terminal when it appears as an autocomplete result.

You should now see a white window with a couple of lines of text. At the end of the last line is a blinking cursor where you can type in commands.

```
Last login: Mon May 16 14:08:02 on ttyp2
Welcome to Darwin!
x820rs116:~ larry$
```

Ubuntu

In Ubuntu, opening the shell varies enormously depending on the particular distribution and window manager you're using. You'll need to determine how to open the shell for your particular version, possibly with a little help from the Internet. Once you open the shell, you should see a window with at least a line of text. At the end of the line is a blinking cursor where you can type in commands.

```
larry ~$
```

2.2.2 Using the Shell

When your shell is ready for interaction, you'll see a character followed by a blinking cursor where you can type in text. The exact character will vary from operating system to operating system. In Windows, for example, you'll usually see some text and a > followed by a blinking underline cursor. In macOS and Ubuntu, you'll usually see some text and a `$` followed by a blinking cursor.

The text to the left of the character will vary from system to system, so throughout this book, we will simplify the shell prompt to just the `$` character.

```
$
```

You can type the command that you want to execute after the `$`. Note that the set of commands that the Windows PowerShell understands is slightly different than the set of commands that the macOS terminal and the Ubuntu shell understand. (The macOS and Ubuntu shells both understand the same set of commands, however.) The commands we teach in this section work identically across all three operating systems with a few limited exceptions that we'll guide you through.

Throughout this book, we'll present the commands that you will type in *italics* to distinguish it from text that the computer prints, like so:

```
$ echo "hello, world!"
```

To execute a command, press enter or return. You will see the computer respond to what you've entered. We will display information written by your computer without italics.

```
$ echo "hello, world!"
hello, world!
```

Checking Your Python Installation

To ensure that Python is installed correctly, enter the following command:

```
$ python3
```

If this command produces an error, try the `python` command, without the `3`. Make a note of the command that worked for you; you will be using it extensively throughout this book.

If Python has installed correctly, you should see something like this:

```
$ python3
Python 3.5.1 (v3.5.1, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on
    win32
Type "help", "copyright", or "license" for more information.
>>>
```

Make sure that the version number next to "Python" on the first line of output begins with a 3, signifying that you are using version 3 of Python. If

it shows a 2, go back to the installation instructions for your system and make sure to install Python 3 instead.

If Python is installed correctly, this command opens the Python interpreter within your shell. You should see >>> with a blinking cursor at the end. Later, you will see that this is a special system called the *Python interactive environment*, which allows you to interactively type and execute Python commands directly into your shell. (We will often refer to the Python interactive environment as interactive Python for the sake of brevity.)

For now, enter the following code and press *enter*. This is how you exit the Python interactive environment:

```
>>> quit()
```

You can think of interactive Python as special shell within the normal shell used by your particular operating system. You enter interactive Python using the `python3` command on your operating system's shell, and you leave interactive Python and return to the normal shell using the `quit()` command. The two environments understand different sets of commands—the shell speaks a language defined by your operating system, while interactive Python speaks Python. Throughout this book, we differentiate between the two environments by beginning commands with `$` on the normal shell and >>> in interactive Python. No matter what operating system you run, interactive Python on your computer will always precede a blinking cursor with >>>, giving you an easy way to determine which environment is currently in use.

Checking Your Pip Installation

Next, you need to check that the program *Pip* has installed correctly. Pip allows you to download libraries of Python code that others have written so you can give Python a whole load of features not included in the basic language, like the ability to interact with websites and PDFs, as one example. To test that you have Pip installed, enter the followlwing command:

```
$ pip3 --version
```

As with the `python3`, command, you may need to enter `pip -version` if you're getting an error. At the end of the output from your computer, you should see the Python version that the command corresponds to. Make sure the version number starts with a 3, and make a note of the command that worked properly.

```
$ pip3 --version
pip 1.5.4 from /usr/lib/python3/dist-packages (python 3.5)
```

2.2.3 Navigating with the Shell

All of the files on your hard drive are organized into *folders*. You probably have a folder called *Documents* to store files for school or work, another folder called

Music to store your songs, and others such as *Pictures* and *Downloads*. Folders on your computer are organized into a hierarchy; your *Documents* folder can contain *subfolders* for specific kinds of documents, like *memos*, or documents for particular purposes, like *hawaii-trip-2016*.

When you need to find a file, you usually start by opening a file explorer (Finder on OS X or Windows Explorer on Windows). A file explorer gives you a graphical way to see the folders and files on your hard drive, navigating deeper into the layers of folders on your computer until you find the file you're looking for. To most computer users, this process is second nature—something you use every day.

We reviewed this seemingly obvious process in such detail for a reason: your shell contains an equivalent, text-based way of exploring the files on your computer, one that we will use extensively throughout this book.

When you open a graphical file explorer, you traverse your hard drive by navigating from folder to folder. At any given moment, you are looking within one particular folder (*Documents*, *Pictures*, or *Memos*). Your shell does the same thing; it always has a current working directory in which it is operating.

To see which folder you are working in, simply enter:

```
$ pwd
```

In macOS and Ubuntu, you will see output like this:

```
$ pwd
/home/larry
```

In Windows PowerShell, you will see:[1]

```
$ pwd
Path
----
C:\Users\Larry
```

These results are called *paths*, showing you that you are in the folder `larry` within the folder `home` (in the first example) or in the folder `Larry` in the folder `Users` on the C-drive (in the second example). In later chapters, we will explore the structure of files and folders in far more detail; our goal here is to give you a basic working knowledge of navigating through your files on the shell.

Your shell has an entire text-based version of the interface you would see in a graphical file explorer. To see a list of all the files and folders in your current working directory, enter the `ls` command. The result on macOS and Ubuntu will look like this:

```
$ ls
Desktop    Downloads Pictures
Documents  Music     brief.doc
```

The result in Windows PowerShell will look something like this:

[1] For historical reasons, macOS and Ubuntu use a forward slash while Windows use a back slash.

$ *ls*

Directory: C:\Users\Larry

Mode	LastWriteTime	Length	Name
d------	05/13/2016 01:19 PM		Desktop
d------	05/13/2016 01:19 PM		Documents
d------	05/13/2016 01:19 PM		Downloads
d------	05/13/2016 01:19 PM		Music
d------	05/13/2016 01:19 PM		Pictures
-a-----	05/16/2016 01:19 PM	1733	brief .doc

Just like in the file explorer, you can navigate deeper into a folder within your current working directory. To do so, enter:

$ *cd folder_name*

with the name of the folder that you wish to enter in place of `folder_name`. (The `cd` command is short for "change directory.") Note that, if the folder name contains spaces, you need to put the name of the folder in double quotes.

In the next example, we use the `pwd` command to determine that we are currently in the `larry` directory within the `home` directory. We then navigate to the `Documents` directory within the `larry` directory and use `pwd` to verify that we are indeed in the new location. In macOS and Ubuntu:

$ *pwd*
/home/larry
$ *cd Documents*
$ *pwd*
/home/larry/Documents

In Windows PowerShell:

$ *pwd*
Path

C:\Users\Larry
$ *cd Documents*
$ *pwd*
Path

C:\Users\Larry\Documents

To navigate up one level, for example, from /home/larry/Documents back to /home/larry, you can enter the command `cd ..` (`cd` followed by a space and two dots). In macOS and Ubuntu:

$ *pwd*
/home/larry/Documents
$ *cd ..*

```
$ pwd
/home/larry
```

In Windows PowerShell:

```
$ pwd
Path
----
C:\Users\Larry\Documents
$ cd ..
$ pwd
Path
----
C:\Users\Larry
```

2.3 IDLE

The last component we need to write Python programs is an *Integrated Development Environment* (IDE); for short, it is often referred to as an *editor*. Just as you use Word to edit documents, IDLE is a program specially designed for editing Python files. It understands the Python language and has a number of features to help you compose your programs. IDLE should have installed as a standalone program on your computer when you installed Python and its associated programs earlier, so you should be able to open it as you would any other program (All Programs in Windows and the Python 3.X subfolder within the Applications folder in OS X).

When you first open IDLE, you should see interactive Python—several lines of text and >>> followed by a cursor. To write a Python program, you will need to open up an editor window rather than the interactive window—a blank document that you can fill with code.

Let's make the editor open by default from now on. Click the `Options` dropdown menu and click `Configure IDLE`. Go to the `General` tab and, under the `Startup Preferences` category, click the radio button next to the label `Open Edit Window`. This will ensure that IDLE defaults to giving you an editor window. Click `Apply` then `ok` to finish the process.

To open up a new editor window, click on the `File` dropdown menu and click on the `New File` button. You will now see a blank document into which you can type text. This is where you will compose your Python programs. To start, enter the following:

```
1 print("hello, world!")
```

Now click on the `File` dropdown and save the Python file somewhere where you won't lose track of it; we'll need to know the exact directory where you saved it in another couple of paragraphs. We recommend you create a subdirectory known as *Projects* within your home folder. This will give you a common place to store all of the Python programs you create throughout this book and beyond.

Navigate to within the *Projects* directory and save the program under the name `hello`. The file will save with the *.py* extension, which signifies that it's a Python file.

2.3.1 Running Python Programs

There are two ways to run Python programs. The first is to run the program through IDLE. The second is run it through the shell. Doing so through IDLE will be convenient early on, but you will prefer to use the shell as you begin to write more complicated programs, so we'll show you both here.

Running Programs in IDLE

Open your Python file in the IDLE editor window by going to `File > Open` and finding the Python file. Otherwise, if you just created and saved a program, it should already be open. Click the `Run` dropdown and select `Run Module`. A window with interactive Python should appear and the following output will display on the last few lines:

```
========== RESTART: C:/users/larry/hello.py
    ================
hello , world!
>>>
```

Your Python program did exactly what you told it to: output the text "hello, world!" If you were to continue working on your program, you could return to the editor window, make some changes, save, and run the program again as desired.

Running Programs from the Shell

The second way to run a Python program is to do so via the shell. Open up a new shell and navigate to the *Projects* folder where you saved your *hello.py* program. (If you need a refresher on navigating using the shell, consult the *Navigating with the Shell* section earlier in this chapter). Once there, enter the following command to execute your program:

```
$ python3 hello.py
hello , world!
```

This command tells your computer to execute the Python program called *hello.py* using the Python interpreter, which knows how to speak Python. Remember that, if you used the `python` command rather than the `python3` command earlier, be sure to make the same substitution here and going forward. Also note that, if the name of your Python file contains spaces, you will need to put it in double quotes.

Chapter 3

Data Types and Expressions

In this chapter, you'll begin to formally interact with Python. Before you write full programs, though, you need to understand the basic building blocks of the Python language. The fundamental units of computation in Python are Expressions, small calculations like adding numbers or combining snippets of text that we can put together to solve bigger problems.

Each of these expressions produces a value - a piece of data like the number `2` or the word `'hello'`. As we'll soon see, Python allows you to manipulate different kinds of data in ways specific to the form of information that it stores. For example, Python lets you calculate with numbers, splice together words to form larger passages of text, and even perform formal logic. In all, Python has five fundamental types of data that we will introduce in this chapter: two types of numbers (those with and without digits after the decimal point - integers and floating point numbers, respectively), one type for text - strings, one type for logic - booleans, and a special type to express the absence of information - none. By the end of this chapter, you will be fluent in each of these data types and the operations for manipulating them, paving the way for you to write full Python programs to process this information.

3.1 Basics of Expressions

Before we dig into the details of different data types and expressions, we need to open interactive Python, which we will use extensively throughout this chapter to experiment with data and expressions. Open up the shell on your computer (the terminal or PowerShell) and enter the command to activate interactive Python. As we discussed in Chapter 2, that command may be python3 or python When you start interactive Python, you should see a prompt similar to the following:

```
$ python3
Python 3.5.1 (v3.1.4, Dec 6 2015 01:38:48) [MSC v.1900 32 bit (Intel)] on
    win32
Type "help", "copyright", or "license" for more information.
>>>
```

Make sure the Python version number that appears when you start interactive Python begins with the number 3. After the >>> you should see a blinking cursor where you can type in text. In its most basic form, interactive Python functions as a calculator. Try out the following code:

```
>>> 2 + 2
4
>>> 372 - 211 + 17
178
```

Each of the commands you entered was an expression—in this case, a numerical expression. Python evaluates the expression, computing the resulting value and displaying it for you to see.

Expressions can process different types of data beyond simple numbers, such as text, files, web pages, and PDF documents. Different types of data often require different operations. For example, we use operations like addition and subtraction on numbers, but we could use operations like finding, replacing, and splicing on expressions that involve text.

As you write programs, you'll need to keep track of the types of data you're working with. If you accidentally mix different types of data in a way Python doesn't understand, like trying to divide a number by some text, your program will crash—terminate unexpectedly due to an error.

Python has five basic data types on which you can operate:

- *Integers.* Whole numbers—numbers with nothing after the decimal point. This is the type that we saw in our last examples.
- *Floating point numbers.* Numbers that are allowed to have something after the decimal point. Later, we'll look at why there are two types of numerical data in Python (and most other programming languages).
- *Booleans.* The logical values True and False. Booleans are convenient for comparing values. Is one number bigger than another? Are two numbers equal? Are two sentences the same?
- *None.* A type representing nothing—the absence of data. Although it might seem extraneous now, we'll show you its many uses in later chapters.
- *Strings.* Words and text—literally "strings of characters."

In the following sections, you'll learn the basics of each of these types and get familiar with the way Python evaluates expressions. By the end of this chapter, you'll have all the pieces you need to begin writing Python programs. Although these types may seem simple at first glance, they can be combined to build far more powerful types, like web pages.

3.2 Integers

First we'll discuss integers: (shorthand int) whole numbers that do not have anything after the decimal point. Much like a calculator, Python will give you access to all of the operations of standard arithmetic. Rest assured that readers with mathematical inhibitions should fear not: numbers are important in Python, but the most complicated math in a typical Python program is counting. Integers are relatively simple and intuitive types, but they'll demonstrate the essential concepts necessary to understand more sophisticated kinds of data.

3.2.1 Basic Arithmetic

Integers include both positive and negative numbers without decimal points. Examples of integers include 0, 112, and −37. If you enter a number into interactive Python, Python will calculate its value and give you back exactly what you put in. This is because, when Python evaluates the expression 112, it unsurprisingly arrives at the value 112.[1]

```
>>> 0
0
>>> 112
112
>>> -37
−37
```

Python includes basic mathematical operators. You can add with the + operator, subtract with the − operator, and multiply with the ∗ operator. Try this in the shell:

```
>>> 3 + 4
7
>>> 27 - 16
11
>>> 8 * 9
72
```

You can also take exponents. That is, you can square numbers (5^2) or take other exponents (3^6 and 7^0). To do so, you would enter something like 5 ∗∗ 2 where the number on the left is the base and the number on the right is the exponent.

```
>>> 5 ** 2
25
>>> 3 ** 6
729
>>> 7 ** 0
```

[1] In fact, this is the technical definition of a *value*: an expression on which no further computation can be performed.

1

Python has three different division operators.Often, we want to divide two integers and get an integer back. However, consider what would happen if we tried to divide 15 by 4: the answer will be 3.75. But 3.75 isn't an integer—it has numbers after the decimal point. When we want to stay exclusively within the world of integers, this rule of math is rather inconvenient.

For instance, suppose you have 45 students going on a field trip and know that each of your buses can hold 18 people. How many buses do you need? The answer isn't 2.5, since half of a bus isn't much use for transport. Python has an integer division operator: // which divides two numbers and discards everything after the decimal point, rounding the result down to the nearest integer. This operator is known as the floored division operator. For our example, you could enter 15 // 4 and Python does the normal division (3.75) and then rounds down to eliminate the decimal, arriving at 3.

```
>>> 15 // 4
3
>>> 3 // 2
1
```

Returning to our bus example, we would write `45 // 18 + 1` to compute the number of buses we need. The first part of the expression (`45 // 18`) computes that we need two buses (since the `//` operator rounds down), taking care of 36 of our students. To cover the remaining 9 students, we add the `+ 1` at the end of the expression. The overall effect of adding the `+ 1` at the end of the expression is to perform integer division but round up.

You need to be especially careful with negative numbers when using the // operator. Because it always rounds down, while 15 // 4 will evaluate to 3, −15 // 4 will evaluate to −4.

```
>>> -15 // 4
−4
```

Sometimes you'll want to know the remainder rather than throwing it away. For example, 2 goes into 5 twice and leaves a remainder of 1. With our field trip example, where we have 45 students and buses that hold 18 people, if you ordered only 2 buses you'd be left with a remainder of 9 upset students and an untold number of upset parents. To compute the remainder, you can use the second division operator, the `%` operator, known as the modulus operator.

```
>>> 5 // 2
2
>>> 5 % 2
1
>>> 14 // 4
3
>>> 14 % 4
2
```

```
>>> 45 // 18
2
>>> 45 % 18
9
```

We'll discuss to the third division operator when we get to floating point numbers, since it requires familiarity with floating point numbers.

3.2.2 Errors

Let's try one more experiment with division: dividing by zero. You may remember that, in math, this operation is considered illegal. How does Python handle it?

```
>>> 4 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This output is Python's cryptic way of saying that it was unable to evaluate the expression, causing the program to crash. The traceback is the error report that Python provides to help you understand why the program crashed. In our case, we have a ZeroDivisionError. Python has a wide range of possible types of errors, of which this is just one. You'll become familiar with plenty of others as you gain experience as a programmer. In this case, Python provides a helpful error message: integer division or modulo by zero. In other words, you tried to divide by zero, but Python doesn't know how to, so it crashed.

Python has different errors for different circumstances. For example, if you tried to add two numbers but accidentally forgot to enter the second one, you'd get an error like this.

```
>>> 2 +
  File "<stdin>", line 1
    2 +
      ^
SyntaxError: invalid syntax
```

This error message is slightly more generic. Python says that it found a syntax error; it could not process the ill-formed code you provided because it wasn't a valid Python expression. Python has helpfully pointed out precisely where the error took place: at the plus sign. By looking closely at the error message, you should be able to figure out why Python struggled to make sense of your expression and how to fix it.

Running into error messages is a natural part of programming. Don't feel that it is in any way an indictment of your aptitude for programming if you see them all the time. Error messages are an important source of feedback that can help you pinpoint mistakes as you develop your programs. Even expert programmers spend the majority of their time revising their code to eliminate

the sorts of bugs that cause crashes. Thankfully, running experiments is very cheap in computer science, so there's no harm in executing your program ten (or a hundred) times until it runs error-free.

From time to time, though, the error messages will seem inscrutable, and you are entitled to feel more than a little frustrated when Python rejects your program for mysterious reasons. It's equally frustrating when you've spent an hour wrestling with an error message only to find that you forgot a punctuation mark somewhere that happened to be important. You will get more comfortable with making sense of error messages and working through the debugging process in a systematic manner as you gain experience. Professional software engineers (including the authors of this book) spend a large fraction of their time tracking down the sources of errors like these.

3.2.3 Larger Expressions and Precedence

Back to integers. You can build larger formulas out of basic arithmetic operators (addition, subtraction, multiplication, and division). You could add several numbers or mix all of the operations together:

```
>>> 1 + 2 + 3 + 4 + 5
15
>>> 12 * 4 - 8 // 3
46
>>> 1 * 2 * 3 * 4 * 5 - 7
113
```

When you have an expression with multiple operators, Python needs to be able to figure out which parts to evaluate first. In other words, it needs to know if:

$3 * 4 - 2$

would equal 10 (multiplying 3 and 4 before subtracting 2) or 6 (subtracting 2 from 4 before multiplying by 3). Thankfully, in programming, the cost of experimenting is small, so the easiest answer is simply to try it.

```
>>> 3 * 4 - 2
10
```

We have our answer, but there are also general rules that Python uses to make these decisions that are useful to know. Each Python operator has a precedence—the priority it gets when Python evaluates an expression. The highest priority operators get evaluated first. For example, in the expression above, Python evaluated the multiplication first because the $*$ operator has higher precedence. Python therefore evaluated the expression $3 * 4$ to get 12 before evaluating the remaining expression $12 - 2$, finally reaching 10.

Below is the order of precedence for the operators you've learned about so far. Items that are higher on the list get evaluated first. This is the same ordering used in standard arithmetic.

1. Exponentiation (∗∗)
2. Multiplication (∗) and division (// and %)
3. Addition (+) and subtraction (−)

When multiple operators have the same precedence, results are evaluated from left to right. For example, let's try to evaluate this expression

2 * 4 + 6 ∗ 8 − 3 ∗ 5

First, Python sees that multiplication is the highest-precedence operator, so looks for multiplications to evaluate first. Since there are multiple multiplications, it looks for the leftmost, which is 2 ∗ 4. It evaluates this expression to 8 and begins again with the formula

8 + **6 * 8** − 3 ∗ 5

The highest-precedence operator left is still multiplication and the leftmost instance is 6 ∗ 8, which it evaluates to 48. Now, the expression is

8 + 48 − **3 * 5**

The final multiplication has the highest precedence, so Python evaluates 3 ∗ 5 to 15.

8 + 48 − 15

With no multiplications left, Python looks for the highest precedence operator. Since addition and subtraction have equal precedence, Python evaluates the leftmost instance of either one. The addition of 8 and 48 evaluates to 56, giving us:

56 - 15

Finally, with only one option left, Python evaluates the expression and arrives at 41. We can confirm this result by typing it into the Python shell.

```
>>> 2 * 4 + 6 * 8 - 3 * 5
41
```

3.2.4 Setting Your Own Precedence with Parentheses

Sometimes you'll want expressions to evaluate in ways that go against Python's default precedence hierarchy. You can direct Python to evaluate certain parts of an expression before others using parentheses, just as you would on paper. For example, if you wanted the subtraction in

3 ∗ 4 − 2

to precede the multiplication, you would write

3 ∗ (4 − 2)

Python will always evaluate expressions within parentheses before evaluating anything outside. In this case Python will evaluate $4 - 2$ to 2, simplifying the larger expression to $3 * 2$, which it can evaluate to 6.

```
>>> 3 * (4 - 2)
6
```

You can put parentheses within parentheses as necessary to control Python's evaluation. Consider our expression from earlier, now with added parentheses:

$2 * (4 + 6 * \mathbf{(8 - 3)}) * 5$

Python will see the parentheses around the expression $4 + 6 * (8 - 3)$ and know to evaluate it first. When it tries to evaluate that expression, it will see another layer of parentheses that it must evaluate first: the subexpression $8 - 3$, which it evaluates to 5.

$2 * (4 + \mathbf{6 * 5}) * 5$

Still within the parentheses, Python will look for the operator with the next highest precedence, and will evaluate the multiplication $6 * 5$ to 30.

$2 * \mathbf{(4 + 30)} * 5$

Seeing one expression left within the parentheses, it evaluates $4 + 30$ to get 34.

$\mathbf{2 * 34} * 5$

Having completed the expressions in parentheses, Python looks for the highest precedence operator remaining. Finding two multiplication operations, it begins by evaluating the leftmost one.

```
68 * 5
```

Finally, Python reaches the answer, 340, by evaluating the final multiplication.

In many cases parentheses are necessary to get Python to evaluate expressions in the order you want. Even when they're not strictly required, however, you should use parentheses whenever you think it may be unclear how Python might evaluate your expression. These extra parentheses can add clarity for someone reading your program. You should always write your code with a reader in mind, and this is one way of doing so. As we discuss a wide variety of operators through the following sections, it will be impossible to keep track of all of the precedence rules in your head. Whenever you don't remember the precedence, use parentheses to ensure Python behaves as you intend.

Style note. It is considered good style to leave one blank space on either side of Python operators. This spaces out the operators and values, turning a jumble of symbols into a legible mathematical expression.

3.3 Floating Point Numbers

Now that you have a solid understanding of the way Python handles integers and evaluates expressions, we can start discussing additional types. The next type we will study is floating point numbers (shorthand: *float*). Floating point numbers are all numbers with a decimal point. Examples of floats include 3.14, 18.42159, and −27.91828. Like integers, floats can be positive or negative.

Even numbers that only have zeros after the decimal point are floating point numbers. For example, the numbers `7.0` and `93.00` are floating point numbers. This may seem confusing at first, because the numbers `7` and `93` are integers. In our day-to-day lives, the numbers `7.0` and `7` are the same. However, Python sees them as two separate entities: a floating point number (`7.0`) and an integer (`7`). As we mentioned at the beginning of this chapter, Python allows us to perform different kinds of operations on different kinds of data. The integer `7` can interact with other integers using the operators we discussed in the previous section, while the float `7.0` can interact with other floats as we'll see shortly. Floats are always written with a decimal point to distinguish them from integers. For example, the number 3 is an integer but the numbers 3.0 and 3.0000 and 3. (just a decimal point with nothing afterward) are all floats. This distinction is subtle but important. Accidentally mixing types can cause Python to crash or even worse—unexpected behavior that causes your program to silently and subtly produce the wrong results–so it's always important to know what type you're dealing with.

It's worth asking why Python has two different kinds of numbers and why we should bother to distinguish integers from all other numbers. The short answer is that there are many circumstances when programs need to count, and counting only makes sense with integers. For example, if we want to download the twenty most recent Supreme Court cases, it makes sense to speak only in terms of whole numbers. You will never have 3.79 Supreme Court cases. Counting comes up so often within programming that it makes sense to create a special category for integers. This distinction is made in nearly every modern programming language.

The addition (+), subtraction (−), multiplication (∗), and exponentiation (∗∗) operators are identical to those for integers. The only difference is that, when evaluated on floating point numbers, they produce a floating point result.

3.3.1 Division with Floats

The two division operators from the last section (`%` for modular arithmetic and `//` for floored division) work the same way on floating point numbers as they do for integers except that they produce floating point results when used on floating point numbers. In addition, we have Python's third division operator, which produces exact, fractional outputs. This operator is written with a single forward slash: `/`. It exactly divides two numbers: `3.0 / 2.0` will evaluate to 1.5.

```
>>> 5.0 / 2.0
```

```
2.5
>>> 14.0 / 4.0
3.5
```

The / operator will always produce a floating point number, even when it is used on two integers.

```
>>> 5 / 2
2.5
>>> 14 / 4
3.5
```

Operator precedence and parentheses behave exactly the same with floating point numbers as with integers.

3.3.2 Mixing Integers and Floating Point Numbers

In practice, you'll often want to mix integers and floats in arithmetic. When you mix the two types, Python automatically converts integers into floats before doing the arithmetic, like so:

```
>>> 9 - 3.1
5.9
>>> 3 * 1.5
4.5
```

This conversion means that the result of multiplying 12 by 2 will be different than the result of multiplying 12 by 2.0; the former will be an integer, while the latter will be a floating point number.

```
>>> 12 * 2
24
>>> 12 * 2.0
24.0
```

3.3.3 Converting Between Number Types

It's also possible to convert an integer into a float without needing to actually do any calculations. To convert the integer 12 into the float 12.0, you would enter `float(12)`. Python converts whatever expression you put in the parentheses into a float. You can also put an entire expression inside the parentheses; just as with other parentheses, Python will evaluate the expression inside before converting it. The parentheses are always required for conversion, though, even when the expression inside is just a single number like 12.

```
>>> float(12)
12.0
>>> float(4 + 3 + 2 + 1)
10.0
```

```
>>> float(-5 * 3)
−15.0
```

You can also use the int command to convert a float into an integer. For example, you would convert the float 12.0 into the integer 12 by entering int (12.0). This works perfectly fine for a number like 12.0 that can be faithfully represented as an integer because it doesn't have anything after the decimal point, but what about floating point numbers that have digits after the decimal point? Python will simply throw away everything after the decimal point.

```
>>> int(12.0)
12
>>> int(2.5)
2
>>> int(-2.5)
−2
```

This process of actively converting between types using commands like `float()` and `int()` is known as type casting.

3.3.4 Infinity

Floating point numbers have two special values: infinity (inf) and negative infinity (−inf), which are the biggest and smallest possible floating point numbers. To create these values, you use the float type-casting function and enter `'inf'` (or `'−inf'` into the parentheses:

```
>>> float('inf')
inf
>>> float('-inf')
−inf
```

These numbers behave as you would expect infinity to behave. Adding one to infinity is still infinity.

```
>>> float('inf') + 1
inf
```

Multiplying infinity by negative one produces negative infinity.

```
>>> float('inf') * -1
−inf
```

Python has these special values for a particular purpose. Occasionally, you need to create a number that you know is bigger (or smaller) than any other number you'll use in your code. For this you'd use infinity and negative infinity. For example, suppose a user provides a list of numbers and you want to keep track of the biggest number you've seen so far. Before you've seen any numbers, what value should you store? Negative infinity, since you know that—no matter what number you see first—it will be bigger than that.

3.3.5 A Word of Warning

It's important to keep in mind that floats aren't always perfectly exact. To understand what this means, consider how your computer actually stores these numbers. Your computer represents numbers as 1's and 0's. Since your computer has a finite amount of memory with which to do so, it has a limited number of 1's and 0's in which it can store these numbers.

However, some numbers, like the result of evaluating 1 / 3, have an infinite number of digits after the decimal place if written precisely (0.33333...). At a certain point, your computer must round off these numbers so that it can store them in a finite amount of memory. This rounding creates or destroys tiny fragments of numbers that can add up to big errors in calculations. Take the numbers $^1/_3$ and $^1/_4$. When we add them together on paper, we get exactly $^7/_{12}$. Let's try that in Python.

```
>>> 1 / 3 + 1 / 4
0.5833333333333333
>>> 7 / 12
0.5833333333333334
```

These two results do not look the same, and if you were to ask Python if they're equal (as we'll learn to do in the next section), it would say that they are not. These minor inconsistencies rarely show up in practice, but it is important to keep this effect in mind. This example provides one other reason why Python distinguishes integers from floating point numbers; all integer math is exact, so you can be certain that these errors will never show up.

3.4 Booleans

Compared to integers and floating point numbers, booleans (shorthand bool) are far simpler. There are only two boolean values, True and False. Booleans allow you to ask questions that compare values to one another. Is 3 greater than 2? True. Is it greater than 4? False. You can also use boolean operators to build larger logical expressions that ask more nuanced questions: Is 3 greater than zero *and* even? False. But is 3 greater than zero *or* even? True.

Boolean values enable your programs to ask questions that are critical for accomplishing almost any task. Did a person type in the correct password? If the answer is True, we should grant her access to the secret files. If the answer is False, we should lock her out. Nearly every meaningful program asks these sorts of questions and adapts its behavior based on the answers. Booleans provide the language in which to do so.

Note that capitalization matters in Python. True is a boolean value, while true is meaningless. Both boolean values begin with a capital letter. As we learn more operations in Python, you will come to see that everything in Python is case-sensitive; capitalization must be correct.

3.4.1 Equality

The most fundamental question you can ask about two values in Python is, "Are they equal?" The double equal sign operator == compares two values for equality. If the values are equal, the expression will evaluate to True; otherwise, it will evaluate to False. For example, 4 == 4 is True but 4 == 3 is False. You can put entire expressions on either side of the == operator; numerical expressions have higher precedence than the == operator, so Python will evaluate both sides before testing for equality.

```
>>> 12 == 12
True
>>> 8 == 9
False
>>> 3 + 4 == 5 + 2
True
```

The == operator works on all types—even booleans. If integers and floats are mixed, Python converts the integers to floats.

```
>>> 3.5 == 7 / 2
True
>>> 3.5 == 3.4
False
>>> 3.5 == 7 // 2
False

>>> 2 == 2.0
True

>>> True == True
True
>>> False == True
False
```

Just as with the arithmetic operators, you can build compound expressions that include parentheses and multiple comparisons. (Take a careful look at the expressions that follow and evaluate each subexpression one by one as if you were the Python shell.)

```
>>> True == (3 == 3)
True
>>> (5 == 3 + 2) == (3 == 4)
False
```

Python also has the != operator, which tests whether two values are *not* equal to one another. If they aren't equal, it evaluates to True; if they are equal, it evaluates to False.

```
>>> 3 != 11
```

```
True
>>> 12 != 12
False
>>> 8 != 9
True
```

The result of using != will always be the opposite of the result of using ==.

You can flip a boolean value to test for its opposite using the not() operator, also called negation. The expression not(True) evaluates to False and not(False) to True. You can also put expressions in the parentheses, like so:

```
>>> not(True)
False
>>> not(False)
True
>>> not(3 == 2 + 1)
False
>>> not(not(not(True)))
False
```

When dealing with boolean operators, you'll quickly find that there are many ways to say the same thing. For example, the expressions not(4 == 3) and 4 != 3 are asking the exact same thing. When faced with such a choice, you should use your discretion about which of the two is more readable; generally, it is the more compact version (in this case, 4 != 3).

3.4.2 Comparisons

Using booleans, you can make comparisons beyond just equality:. you can also ask whether one number is bigger than another. Python has four operators for doing so.

The first two comparison operators are > and <. The > operator asks whether the number on the left is bigger than the number on the right. For example, the expression 4 > 3 evaluates to True (since 4 is bigger than 3), but 2 > 3 evaluates to False (since 2 is not bigger than 3).

You can use the < operator to ask precisely the same question the other way around; is the number on the right bigger than the number on the left? As with the other boolean operators, you can also place expressions to the left and right of these operators; the comparison operators have lower precedence than the numerical operators, so numerical expressions will be evaluated first.

```
>>> 4 > 3
True
>>> 4 < 3
False
>>> 2 > 3
False
>>> 2 < 3
```

True

```
>>> (3 + 1) * 5 > 14 + 3
True
>>> 5 + 6 < 3 * 3
False
```

If Python tries to evaluate two equal values with a < or > (for example, `4 > 4`) the expression will evaluate to False.

```
>>> 4 > 4
False
>>> 8 < 8
False
```

The two other comparison operators are >=, which checks whether the left number is greater than *or equal to* the right number, and <=, which checks whether the left number is less than *or equal to* the right number. So, although 3 > 3 evaluates to False, 3 >= 3 evaluates to True. If the two values are not equal, the >= and <= operators function identically to > and <.

```
>>> 5 >= 4
True
>>> 5 > 4 + 1
False
>>> 5 >= 4 + 1
True
```

3.4.3 Compound Operators

Often, you'll want to test more than one condition at the same time. For example, you may want to check whether the numbers 7, 8, and 6 are in order from least to greatest. To do so, you need to perform multiple comparisons. First, you'd check that $7 < 8$, which is True. Then you'd check that $8 < 6$, which is False. Finally, you need to somehow merge these pieces of information together into a single boolean conclusion about whether $7 < 8$ and $8 < 6$, which would be sufficient to decide whether the numbers 7, 8, and 6 are in ascending order.

The Python and operator combines boolean values together in this fashion. It evaluates to True if the two values are both True; otherwise, it evaluates to False. In other words, the expression a and b is True if the expression a is True and, separately, the expression b is True.

As another example of the behavior of the `and` operator, consider the the following situation: you will only walk to work if (1) it is sunny outside `and` (2) you woke up on time. Both of these conditions are boolean expressions. Is it sunny outside? `True`. Did you wake up on time? `True`. Since both of these conditions are true, then you will indeed walk to work (`True`). However, if either

or both of these conditions are `False`—if it isn't sunny outside (condition (1) is `False`) or you didn't wake up on time (condition (2) is `False`) or both (it isn't sunny and you didn't wake up on time)—then you won't walk to work: the combined expression (1) `and` (2) is `False`. This logic is how the Python `and` operator behaves—it only evaluates to `True` if the expressions to its left and right are both `True`.

Below are all the possible permutations of inputs to the and operator, known as a truth table. Since the and operator works on two values and only two boolean values exist (True and False), there are only four possible combinations.

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

When you build boolean expressions, it can be useful to create a truth table to help you reason through how they operate.

To solve the original problem about the numbers 7, 8, and 6, you could enter:

$7 < 8$ and $8 < 6$

Python will first evaluate the comparisons to the left and right, finding that 7 is smaller than 8 but 8 is not smaller than 6.

True and False

Since one of the resulting values is False, the whole expression evaluates to False.

False

You can also use the and operator multiple times within one expression. For example, to test whether three conditions all hold true, you can write

True and True and False

Python will evaluate the first and, finding that it is True since both of its values are True. This leaves the expression

True and False

which you already know evaluates to False.

Python also provides the or operator, which evaluates to True if one or both of its expressions are True. This means that the expression $7 < 8$ or $8 < 6$ is True, since the left expression is True. The truth table for the or operator is as follows:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

As another example of the `or` operator, consider the following scenario: you will wear a jacket if (1) it is cold outside `or` (2) it is raining. If either or both of those conditions are `True`, you will wear a jacket. The only case in which you won't wear a jacket is if it isn't cold outside (condition (1) is `False`) and it isn't raining (condition (2) is `False`).

The and and or operators have the same precedence, so it is important to use parentheses carefully when you mix the two.

There are numerous ways to mix and match and, or, and not to produce equivalent expressions. For example, the expression not(a and b) is the same as not(a) or not(b). As usual, choose the route that is the most readable.

3.4.4 Mixing Booleans with Other Types

Strangely, when you mix boolean and numeric types, True is converted to the integer value 1 and False is converted to the integer value 0. In practice, you should never use True and False in this fashion; it is considered bad programming style. We are only showing you this behavior so that you can recognize when it leads to errors in your code.

```
>>> True == 1
True
>>> True == 0
False

>>> False == 1
False
>>> False == 0
True

>>> 2 + True
3
>>> 2 + False
2
>>> True + True + True + True
4
```

For emphasis, you should *never* use booleans in this fashion, but you may run into subtle errors where you accidentally mixed types in a way that caused this strange behavior to kick in.

3.5 The None Type

The None type is even simple than our booleans—it has only has a single value: None.

```
>>> None
None
```

In Python, None signifies the absence of data—that nothing is there. This may initially seem superfluous, but in practice it is exceedingly valuable. As an example, suppose you're trying to download a web page (something we will show you how to do later in this book). Unfortunately, your computer isn't connected to the internet, so Python fails to connect to the website. What value should it give you back instead? `None`, the absence of information. Even if Python is indeed able to connect, the web page might take a long time to download. What value should you store while you wait for this to happen? Nothing—`None`. As we will see in later chapters, `None` is often a convenient placeholder while you wait for other information or to signify that no information was found.

The only thing you can do with None is test whether something else is also None. This operation, although simple, is vitally important in the sorts of situations just described. Most of the time, Python will be able to connect to the website and download the webpage you requested. But sometimes, it will fail and give you `None` instead. You will want to tell the difference between these two situations so that you can perform different actions depending on the outcome. If the webpage downloaded successfully, then you can further process the data you received. If the webpage did not download successfully (meaning you received `None` instead, you should display a polite failure message to your user; if you tried to process `None` further as if it were a webpage, your program is liable to crash.

To test whether a value is `None`, use the `is` operator. When we put None to the right of the operator, it asks whether the value to the left is also None, and evaluates to the boolean True if it is and False otherwise. For example:

```
>>> 7 is None
False
>>> None is None
True
```

The expression `7 is None` asks "Is the integer 7 the same as the value None?" Since it's not, this expression evaluates to False. But the expression `None is None` evaluates to True.

Python also provides the `is not` operator, which returns the opposite of `is`; one returns True whenever the other returns False.

```
>>> 7 is not None
True
>>> None is not None
False
```

Note that it's considered bad Python style to test whether something is equal to None using the == and != operators. Instead, the is and is not operators should always be used.

3.6 Strings

The final fundamental type of data in Python is strings (shorthand str). In this book, strings will be the most interesting and most important type. Strings are Python's way of representing text—everything from a single letter to a word or a Supreme Court opinion. You create a string by enclosing the text within single quotes. (You can also enclose a string within double quotes, but we will prefer single quotes in this book.)

```
>>> 'a'
'a'
>>> 'hello'
'hello'
>>> 'four score and seven years ago...'
'four score and seven years ago ...'
```

If you forget the quote marks, Python will provide you with a handy error to help you fix your mistake:

```
>>> hello world
  File "<stdin>", line 1
    hello world
              ^
SyntaxError: invalid syntax
```

Python rejects the expression with a syntax error, which is Python's generic way of indicating that a program was somehow ill-formed. If you check the spot that the error indicates is wrong, you can often surmise the mistake.

If you were to start typing a string but forget the closing quote, you'll get a more specific error.

```
>>> 'hello world
  File "<stdin>", line 1
    'hello world
                ^
SyntaxError: EOL while scanning string literal
```

This error message is more helpful. Python has seen the opening quote character, and so surmises that you intend to create a string. It begins scanning along the rest of the expression looking for the corresponding closing quote so it knows where the string ends, but reaches the end of the line (EOL) of text without seeing one. The term string literal refers to the string itself—the text enclosed in quotes. Putting all of this jargon back into English, Python was looking for the closing quote of the string and couldn't find it, causing an error.

3.6.1 Concatenation and Type Casting

Python provides a wide range of operators for manipulating text. We discuss the basic operations here, and in Chapter 8 we'll dig into the rest in more detail.

The most common string operation in Python is the merging of multiple strings together into a single entity. For example, say we wanted to combine the strings 'John' and 'Marshall' together into the name 'John Marshall'. To do that, we would use the + operator.

This may seem counterintuitive at first; doesn't + add numbers together? Operators have different meanings for different types of data. When you give the + operator two integers or floating point numbers, it adds them together. When you give it two strings, it concatenates them, which is the technical term for merging them together. This is yet another reason to be mindful of the types of data you are working with. Let's create our name:

```
>>> 'John' + 'Marshall'
'JohnMarshall'
```

As the output shows, you need to be very careful about spaces when you manipulate strings. Since we didn't include a space at the end of the word 'John' or the beginning of the word 'Marshall', Python obediently merged them together as they were, creating the string 'JohnMarshall'. We'll add a space; it doesn't matter whether you add the space at the end of one string or start of the other:

```
>>> 'John ' + 'Marshall'
'John Marshall'
>>> 'John' + ' Marshall'
'John Marshall'
```

Alternatively, you can also add the space in a separate string as part of a second concatenation.

```
>>> 'John' + ' ' + 'Marshall'
'John Marshall'
```

This is useful if you're working with already-existing strings whose value you don't want to change. Note that a string with one space inside (`' '`) is different from a string with nothing inside (`''`). Although a single space looks blank, it is still a character's worth of text. In contrast, a string with nothing inside (called the *empty string*) literally contains no characters.

Although it may seem pointless to create a string that literally contains nothing, you will find yourself doing so frequently as you become more familiar with Python. The empty string is similar to the number zero in many ways. When you want to add several numbers together, you need to start counting at zero. Likewise, when you want to concatenate several strings together, you start with the empty string and build from there. Like zero, concatenating the empty string to another string does not have any effect.

```
>>> '' + 'John'
```

```
'John'
>>> 'John' + ''
'John'
```

3.6.2 Converting Other Types into Strings

Often, you will want to concatenate strings with values of other types. For example, suppose you know that Larry scored an 89 on his criminal procedure test. You want to somehow combine the string 'Larry' and the integer 89 together into one string. At first glance, you'll be tempted to simply use the + operator.

```
>>> 'Larry got an ' + 89
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The expression crashed because Python can only concatenate strings to other strings, but it found an integer, so it couldn't complete the operation you requested. Instead, you'll need to convert the integer 89 into the string '89'. The str operator converts the argument you pass it into a string.

```
>>> 'Larry got an ' + str(89)
'Larry got an 89'
```

This type casting is exactly how we converted between integers and floats using the int() and float() operators. Each basic type has its own type casting command. The str() operator can be used on most types that you will encounter in Python. As always, Python evaluates the expression within the parentheses before performing the type conversion.

```
>>> str(27)
'27'
>>> str(8 + 7)
'15'
>>> str(3.14)
'3.14'
>>> str(True and False)
'False'
>>> str(int(27.2))
'27'
```

Similarly, some strings can be converted into other types, as long as they're formatted properly.

```
>>> int('1234')
1234
>>> float('3.14')
3.14
```

```
>>> bool('True')
True
```

Of course, when converting types, you must be wary of errors. Not all strings can be converted into any type. For example, if you try to convert a string like 'John Marshall' into an integer, you'll get an errror:

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

This Python error is actually quite clear; the string 'John Marshall' isn't a valid integer, so using the int() operator caused the expression to crash. You must make sure your string contains a valid value of the resulting type if you want to convert it.

3.6.3 Replication with *

The + operator isn't the only mathematical operator that can be used for strings. You can also replicate a string using the $*$ operator. For example, if you wanted to create a string with the word 'hi' repeated five times over, you could enter 'hi' $*$ 5 or 5 $*$ 'hi'.

```
>>> 'hi' * 5
'hihihihihi'
>>> 5 * 'bye'
'byebyebyebyebye'
```

As usual, enter spaces where it makes sense for the output.

3.6.4 Length

You can use the len() operator to count the number of characters in a string. For example, to determine the number of characters in the string 'hello', you would enter len('hello'), which will evaluate to the integer 5. The len() operator produces an integer that you can then use in numerical computation. Spaces count toward the length of a string, so the string 'hi there' has eight characters.

```
>>> len('John')
4
>>> len(' ')
1
>>> len('')
0
>>> len('hello' * 5)
25
```

In these examples, the `len()` operator may seem a little superfluous since we already know the length of the string `'hello'`. However, we will often deal

with unfamiliar strings provided by a user or stored in a file, in which case we will need the `len()` operator to evaluate their length.

3.6.5 Single and Double Quotes

So far we've used single quotes to designate strings, though in practice you can also use double quotes. The strings `'hello'` and `"hello"` are identical as far as Python is concerned. The choice of which to use is entirely up to you; just make sure you are consistent. For that reason, we'll continue to use single quotes to define strings in this book.

Having two types of quotes is particularly useful when you need to include a quote in the string itself. For example, using only single quotes, it would be impossible to include the word can't, since Python would confuse the apostrophe for the closing quote of the string. In these cases We can use double quotes to create the string `"can't"`. We'll look at this topic more deeply in Chapter 8.

3.7 Conclusion

We have covered an enormous amount of ground in this chapter. You now understand what Python code looks like and how your computer turns the code you type into meaningful computation. You wrote your first fragments of Python programs, creating and evaluating a wide variety of expressions in all of Python's fundamental types of data. You learned arithmetic, boolean, and string operators and encountered your first Python error messages. This foundation will provide you with everything you need to begin writing programs. In the next chapter, we'll leave interactive Python, transition into the IDLE development environment, and learn how to write entire programs that accomplish appreciable work.

3.8 Cheatsheet

Types

Type	Abbr.	Description	Examples
Integer	`int`	Positive and negative numbers with nothing after the decimal point.	`12`, `8`, `1`, `0`, `-1`, `-18`
Floating Point	`float`	Positive and negative numbers with or without something after the decimal point.	`12.6`, `9.0`, `5.33`, `0.0`, `-3.14159`
Boolean	`bool`	Whether something is true or false.	`True`, `False`
None	n/a	The absence of data.	`None`
Strings	`str`	Text, including characters, words, sentences, special characters, etc.	`'c'`, `'hello'`, `''`, `'1337 h4x0r'`

Arithmetic Operators

Op.	Name	Description	Example
`+`	Addition	Adds the two numbers together.*	`3 + 4` evaluates to `7`
`-`	Subtraction	Subtracts the right number from the left number.*	`8 - 6` evaluates to `2`
`*`	Multiplication	Multiplies the two numbers.*	`3 * 6` evaluates to `18`
`**`	Exponentiation	Takes the left number to the power of the right number.*	`2 ** 3` evaluates to `8`
`//`	Integer Division	Divides the left number by the right number and drops anything after the decimal place. Always evaluates to an `int`.	`9 // 2` evaluates to `4`
`%`	Remainder	The remainder of dividing the left number by the right number. Always evaluates to an `int`.	`9 % 2` evaluates to `1`
`/`	Standard Division	Divides the left number by the right number and preserves everything after the decimal place.*	`9 / 2` evaluates to `4.5`

*: If at least one operand is a `float`, the result is a `float`. Otherwise, the result is an `int`.

String Operators (So Far)

Op.	Name	Description	Example
`+`	Concatenation	Joins two string operands into a single string.	`'aaa'+ 'bbb'` evaluates to `'aaabbb'`
`*`	Replication	Concatenates the string operand to itself the number of times specified by the integer operand.	`5 * 'c'` evaluates to `'ccccc'`
`len()`	Length	Evaluates to the integer number of characters in the string in parentheses.	`len('hello')` evaluates to `5`

Boolean Operators

Op.	Name	Description	Example
`==`	Equality	Evaluates to `True` if the left operand evaluates to the same value as the right operand.[1]	`3 == 2 + 1` evaluates to `True`
`!=`	Disequality	Evaluates to `True` if the left operand evaluates to a different value than the right operand.[1]	`5 != 2 + 2` evaluates to `True`
`<`	Less Than	Evaluates to `True` if the left operand is smaller than the right operand.[2]	`3 < 4` evaluates to `True`
`>`	Greater Than	Evaluates to `True` if the left operand is bigger than the right operand.[2]	`5 > 2` evaluates to `True`
`<=`	Less Than or Equal To	Evaluates to `True` if the left operand is smaller than or equal to the right operand.[2]	`3 <= 4` and `4 <= 4` evaluate to `True`
`>=`	Greater Than or Equal To	Evaluates to `True` if the left operand is bigger than or equal to the right operand.[2]	`5 >= 2` and `4 >= 4` evaluate to `True`
`not()`	Negation	Evaluates to `True` (resp. `False`) when the expression in parentheses evaluates to `False` (resp. `True`).[3]	`not(False)` evaluates to `True`. `not(True)` evaluates to `False`.
`and`	Conjunction	Evaluates to `True` when both operands evaluate to `True`.[3]	`True and True` evaluates to `True`. `True and False` evaluates to `False`.
`or`	Disjunction	Evaluates to `True` when at least one operand evaluates to `True`.[3]	`True or False` evaluates to `True`. `False or False` evaluates to `False`.

1: Works on any two operands with the same type.

2: Works on any two operands that can be put into an order from biggest to smallest. Clearly, numbers (`int` and `float`) can be put into such an order. Python also puts strings into alphabetical order.

3: Only works on boolean operands.

Type Conversions

Op.	Description	Example
`int()`	Converts the expression inside the parentheses into an `int`.	`int('1745')` and `int(1745.236)` evaluate to `1745`
`float()`	Converts the expression inside the parentheses into a `float`.	`float('3.14')` evaluates to `3.14`. `float(18)` evaluates to `18.0`
`str()`	Converts the expression inside the parentheses into a `string`.	`str(2)` evaluates to `'2'`. `str(3.14)` evaluates to `'3.14'`

Order of Operations

The operators below are listed in order from highest precedence to lowest precedence. If Python encounters the operations in an expression, it will evaluate those with higher precedence first. If two operators in an expression have the same precedence, Python evalutes them from left to right. You can alter the order of operations by inserting parentheses.

1. `**`
2. `*`, `//`, `/`, `%`
3. `+`, `-`
4. `is`, `is not`, `<`, `>`, `<=`, `>=`, `!=`, `==`
5. `not`
6. `and`
7. `or`

Keywords

Casting—Forcibly converting values of one type into values of another. Examples: using the int(), float(), or str() operators to convert the expression in parentheses into an integer, floating point number, or string (respectively).

Crash—To fail to evaluate properly. A Python expression or program is said to crash when Python is unable to continue to evaluating it. Crashes are caused by programming mistakes, such as providing an operator with the wrong types or trying to divide by 0.

Evaluate—To perform the computation described by an expression. Evaluation completes when an expression has been turned into a value that can be evaluated no further. Example: the expression 3 + 2 + 1 evaluates to the expression 5 + 1, which evaluates to the value 6.

Expression—A series of values and operators combined together to describe a computation.

Operand—The data that an operator manipulates. Example: the numbers added together by the + operator.

Operator—A manipulation of one or more pieces of data to create new data. Examples: adding two numbers, concatenating two strings, testing whether one number is bigger than another.

Precedence—The level of priority that Python gives to each operator. When an operator has higher precedence, Python will evaluate it first. Example: in the expression $3 + 4 * 2$, Python will multiply 4 and 2 before performing addition because multiplication has higher precedence than addition.

Syntax error—When the contents of an expression or program is ill-formed, preventing Python from making enough sense of your program to even attempt to evaluate it. Examples: forgetting the closing quote on a string, forgetting to type the second number when trying to perform addition.

Traceback—The message that Python displays after a program crashes. It describes the precise expression where a program to crash and the specific error that caused it to crash.

Type—The kind of information that a piece of data stores. The type of a piece of data defines the kinds of operations that can be performed on it. Examples: integers, floating point numbers, booleans, and strings.

Value—An expression that cannot be evaluated any further. Examples: 8, True, or `'hello'`.

Chapter 4

Programming Basics

Now that you're familiar with the basic structure of Python expressions, you can begin to design real programs that combine several (and later, hundreds of) expressions together to accomplish a bigger task. We'll start small—converting between Fahrenheit and Celsius and calculating the amount to bill a client—and then scale these building blocks up into much bigger programs over the coming chapters.

We'll begin by discussing variables, which we can use store the results of expressions for later use. This makes it possible to string together extended calculations that start with some input data and follow a number of intermediate steps to compute a result. For example, we might take thousands of comments on a contentious proposed rule at the FCC, for example and form a summary of the comments containing popular words, phrases, and topics.

Programs are most helpful when they can actually ingest data from the outside world, do something with it, and display a result. For this, we need basic input and output syntax. These three concepts—variables, input, and output—form the substance of every useful computer program you will ever write. We introduce all three in this chapter. By the time this chapter is complete, you will have enough knowledge to perform substantive computations with Python.

4.1 Variables

4.1.1 Creating Variables

Until now, you've just worked with small expressions like `2 + 2` and `'hello ' + 'world'`. These expressions are very short-lived; once they're evaluated, the results are displayed but aren't stored for reuse later. Without this ability it's impossible to tackle larger problems that require multiple steps and more intricate calculations.

We can choose to store that data long after it's calculated using variables. Variables are simply names we use to refer to some saved data throughout a

program. For example, we might use y to refer to the value 2; every time we called y, our program would use the associated value 2.

To save a value to a variable, use the = operator (a single equals sign, in contrast to the boolean comparison operator ==). Put the name you want to give the variable to the left of = and the expression or value you wish to store to the right. In other words, you define a variable by assigning it a value with the = operator.

```
>>> y = 2
```

Note: Notice that interactive Python didn't echo any information back. This means the expression y = 2 did not evaluate to a value, but silently stored the result of the expression on the right in the variable y. Expressions that don't produce values are called statements.

Figure 4-X gives a visual representation of what you get when you run the above code:

This box represents the space in your computer's memory that the y variable is given. When you execute the command x = 2, you instruct Python to reserve space in memory designated for x and fill it with the number 2. We'll use diagrams like this one extensively in the coming chapters to help you develop a mental model for the way Python stores and moves data.

You can give variables almost any name you like, with only a few limitations. The names can be as simple as a single character, like y, or a series of highly descriptive characters, like number_of_fcc_comments. Variable names can include upper and lower case characters, numbers, and underscores, but a variable name cannot begin with a number or include a space. You'll find a much longer discussion of variable naming rules and best practices at the end of this section.

4.1.2 Using Variables

To access the value stored in y, enter y. Python evaluates the variable by substituting the value that it stores. In this case, the expression y simply evaluates to 2.

```
>>> y
2
```

Once defined, you are free to use a variable wherever you like, and Python will dutifully substitute in the value it stores. Consider several examples below.

```
>>> y + 3
5
```

This substitution leads to the expression 2 + 3 and is evaluated to the integer 5. Python follows the same pattern for larger expressions.

```
>>> 5 * (y + 3) - y
23
```

Just as before, Python substituted 2 for all appearances of y, creating the expression $5 * (2 + 3) - 2$, which evaluates to 23. Since y stores an integer, you can use it anywhere that you would usually use an integer expression.

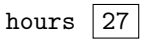
```
>>> 'The value of y is ' + str(y)
'The value of y is 2'
```

Here, Python substituted 2 for y, casting it to a string and concatenating it with the string that preceded it to create the resulting string.

The value of a variable doesn't always have to be a value, either. You can store entire expressions, like so:

```
>>> hours = 11 + 8 + 2 + 6
```

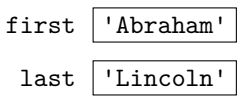
Python will evaluate the expression to the right of = *before* storing it in the variable. When you check the variable hours, you should see the result 27 . Since Python only stores the result of the expression, in memory, a variable made from an expression and a variable made from a single value look the same:



You can create as many variables as you like, and Python gives each unique variable name its own separate storage space.

```
>>> first = 'Abraham'
>>> last = 'Lincoln'
```

In this example, we create two variables. First, we assign the variable first to the string 'Abraham'. Next, we assign the variable last to the string 'Lincoln'. Python sets aside space for each of these variables and fills in the space with the value they store.



We can then use these variables in new expressions.

```
>>> 'Whatever you are, be a good one. -' + first + ' ' + last
'Whatever you are, be a good one −Abraham Lincoln'
```

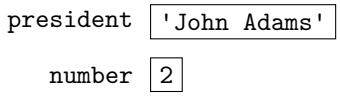
Just as before, Python replaces instances of variables with the values they store.

4.1.3 Reassigning Variables

If you try to assign a value to a variable that already exists, Python won't give the variable new storage space, but will reassign the variable, overwriting the current value with the new value. Consider the example below. First, we create two variables.

```
>>> president = 'John Adams'
>>> number = 2
```

The variable president stores the string name of a president. The variable number stores an integer representing the order in which the president served. Visually:



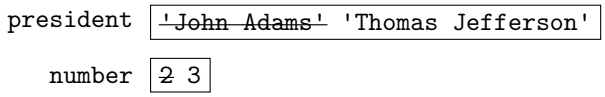
We can then use these two variables to create a string that displays this information.

```
>>> 'President ' + str(number) + ' was ' + president
'President 2 was John Adams'
```

Next, we'd like to update these variables to represent the third president. We can overwrite the old values with new ones using the = operator.

```
>>> president = 'Thomas Jefferson'
>>> number = 3
```

Python has now replaced the previous values stored in the variables with the new values. Visually:



We can then use these updated values just as before, executing the exact same statement with the new variable values.

```
>>> 'President ' + str(number) + ' was ' + president
'President 3 was Thomas Jefferson'
```

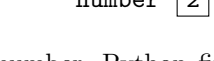
You might notice a pattern here: as we repeat this process for each president, the variable number should increase by exactly 1 each time. Rather than manually writing number = 4 and number = 5 for each subsequent president, we could repeatedly enter

```
>>> number = number + 1
```

If the value of number was previously 3, it would now be 4. If it were 4, it would now be 5. We can redefine the variable number by making it reference to itself. That may initially seem like a contradiction—how can we use the value of number while we're still in the process of defining it?

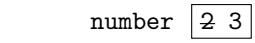
Let's take a closer look at how Python evaluates this expression.

1. Initially, the variable number was assigned some value, let's say 2.



2. When we redefine number, Python first evaluates the expression to the right of = and finds that the value of the variable number is 2. It adds 1, the value to the right of the +, to the old value of number, calculating that the expression number + 1 equals 3.

3. Using this result, it overwrites the old value stored in number with 3, the new value.



In other words, there was no paradox in defining number using itself; we are simply asking Python to calculate the new value of number using its old value. The statement number = number + 1 increases the value of the variable number by 1 each time it's evaluated.

The Self-Reference Pattern

What if we had tried to do this with a variable that hasn't yet been defined?

```
>>> z = z + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

Python produces an error message. Since z hadn't been assigned a value when Python went to evaluate the right side of the expression, Python crashed. The same thing happens if you try to use any variable that hasn't been assigned a value.

```
>>> not_yet_assigned
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'notYetAssigned' is not defined
```

This pattern of self-reference is exceedingly common in Python. For example, you will often find it useful to keep track of the number of words in a string or the number of Supreme Court opinions in a year.

Since programmers love to maximize efficiency (read: do less work), Python has special assignment operators that save a few keystrokes when you want to use this pattern. You can enter this statement, which is equivalent to count = count + 1.

```
count += 1
```

This assignment operator sets count equal to the sum of the current value of count and the expression on the right. The statement

```
count += 7
```

adds 7 to the current value of count, for example. If count was initially 5, it would now be 12. Python includes one of these special assignment operators for each mathematical operator. So the statement

```
count -= 5
```

subtracts 5 from the current value of count and

```
count *= 3
```

sets count to triple its previous value.

4.1.4 Looking Under the Hood

All of the basic types of Python data (integers, floating point numbers, booleans, none, and strings) are stored in variables by value. That means that, if we store a variable inside another variable, only the value is copied over. To illustrate how this concept works, consider the code below.

```
>>> x = 1
>>> y = x
>>> x = 2
```

After executing each of these statements, what is the value of y? We initially set the value of x to 1. Then we set the value of y to x. Because x is an expression, Python evaluates it before storing the result, so y should equal 1 as well.

The question her is whether, when we reassigned x to 2, the value of y change as well. Let's see:

```
>>> y
1
```

Only *the value* stored in x was copied into y; the values were stored in two different places, as the box diagram makes obvious. When we overwrote the value of x, only x was changed, but the place where the value of y was stored was unchanged.

This is how all basic types are handled in Python This point may seem obvious now, but in later chapters we will learn about other types that behave differently—where changing x would have changed y.

As a test of your understanding of variables, suppose you initialized j and klike so: .

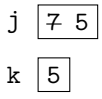
```
>>> j = 7
>>> k = 5
```

Your goal is to swap their values, so that j stores k's value and vice versa. As a first attempt, you might try copying k's value into j and then copying j's value into k.

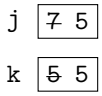
```
>>> j = k
>>> k = j
```

The problem is that when you copy k's value into j, you overwrite j's value and its value is lost forever. Figure 4-x shows where the variables were initially set.

After executing j = k, the two variables have the same value, as shown in Figure 4-x.

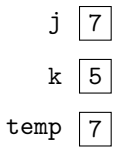


When you then execute k = j, the value 5 is copied back to k.



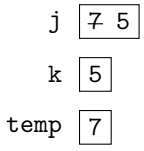
In the end, all you've done is overwrite j with k. To get around this problem, programmers will temporarily store one of the variable in a third variable, to retain its value.

```
>>> j = 7
>>> k = 5
>>> temp = j
```



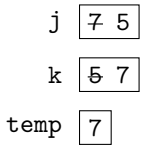
Now you can safely overwrite j with k's value.

```
>>> j = 7
>>> k = 5
>>> temp = j
>>> j = k
```



Finally, you can move the old value of j, stored in temp, into k.

```
>>> j = 7
>>> k = 5
>>> temp = j
>>> j = k
>>> k = temp
```



The two variables have now successfully been swapped. This is a useful trick to remember; you'll run into this situation frequently in practice.

4.1.5 Naming Variables

You should always give your variables descriptive names that explain the information they store. For example, you should use the variable name day_of_the_week , rather than d, so that you or someone else reading your code will have an easier time making sense of it. You should also avoid short variable names like x and y wherever possible. A good rule of thumb is that a variable name's length should

be in proportion to its importance in your program.

Python variable names must start with a letter and can contain letters, numbers, and underscores. For example, thing1 and thing2 are valid variable names but 1stThing and 2ndThing are not. Variables are case sensitive, meaning that Larry is a different variable than larry.

Python has a set of reserved words that you are not allowed to use as variable names becuase they serve other important purposes in Python. Using them as variable names would confuse the Python interpreter. A full list of the reserved words is shown here. Several of these words will already be familiar to you as names with special meaning in Python; we'll come across the others in later chapters.

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

Throughout this book we'll name our variables according to the official Python style guide (known as *Python Enhancement Protocol* 8, or *PEP8* for short). According to PEP8, variables should be named using underscore style, meaning that a variable name consisting of multiple words should be all lowercase and separated by underscores. For example: president_of_the_us and ruth_bader_ginsburg. The underscore style is just one of many popular styles for naming variables. The most important quality in variable naming, as in much of programming, is consistency—use the same naming scheme everywhere—and readability—make sure your names make sense to others.

4.2 Printing Output

At this point, we've done as much as we need using interactive Python. Although we'll return to it occasionally throughout the remainder of the book, it's time to start writing real programs, so you'll need to switch to IDLE. If you need a refresher on opening and using IDLE and the command line, look at Chapter 3. To follow along with the examples in this section (and going forward), you should open a new IDLE editor window and save the file with a memorable name. Be sure to save it somewhere on your computer that you will be able to find again later. You should also open the shell (terminal, PowerShell, etc.) and navigate to the directory where you have saved the Python file.

4.2.1 Writing Python Programs

Now that you have left interactive Python, you can begin writing programs that contain many expressions and statements. A Python program is executed line by line from top to bottom, as if you were entering the commands in interactive Python. If you define a variable on a particular line, you can make reference to it on any future lines.

Unlike interactive Python, Python programs do not echo back the results of evaluating expressions. Instead, you need to explicitly tell Python to print output to the terminal. To do so, we use the print command, followed by a pair of parentheses containing the string we want to output.

For example, you can type the command below into your editor window on the first line and save the document.

```
1 print('Hello, world!')
```

Return to the shell and enter in the following command to execute your program.

```
$ python3 hello.py
Hello, world!
```

Python visited each line of this program (since there was only one line, that means it visited the first one) and executed the statements it found, one by one, from top to bottom.

Python programs can contain multiple lines, so you can return to your editor window and add more print statements. When you execute the code on the left, you will see three lines of output as on the right.

```
1 print('Hello, world!')
2 print('Second line of code!')
3 print('Third line of code!')
```

```
$ python3 hello.py
Hello, world!
Second line of code!
Third line of code!
```

Just as with interactive Python, you can add variables.

```
1  first = 'John'
2  m = 'F'
3  last = 'Kennedy'
4  print(first + ' ' + m + ' ' + last)
5
6  num = len(first) + len(m) + len(last)
7  print('Length: ' + str(num))
```

```
$ python3 hello.py
John F Kennedy
Length: 12
```

Likewise, you can perform calculations.

```
1 deg_f = 70
2 deg_c = int((deg_f - 32) * 5 / 9)
3 print(str(deg_f) + ' fahrenheit')
4 print(str(deg_c) + ' celsius')
```

```
$ python3 hello.py
70 fahrenheit
21 celsius
```

4.2.2 The print Function

The print command, known as a function in Python parlance, told Python to output the string within the parentheses onto the terminal.

You will learn more about functions in Chapter 11. For now, you need to know that a function is a bundle of Python code that you can execute by invoking its name—or, in technical terms, calling it. Functions accept arguments, expressions that you can pass into the function so that the function can use them. The `print` function, for example, accepts a string as input. You could have written any string expression there and watched it appear as output.

The parentheses following a function call are mandatory; they signify to Python that you are calling a function. Some functions take multiple arguments, which are separated by commas within the parentheses in a set order. When you call a function you tell Python to execute the function's code (with the argument as input) and then return back to the current place in the program to continue executing the code that you have written.

The print function is part of the Python standard library, a collection of hundreds of functions built into Python. The standard library contains code that is used so often that a professional has written it and wrapped it in an easy-to-access package. We have already seen several other functions that are members of the standard library. These include the type-casting functions `int`, `float`, `str`, and `bool` and the `len` function, which computes a string's length.

You don't need to see the print function's code to understand how to use it. In fact, it's generally better that you don't see the code. As long as you know how to call it correctly, and it works properly whenever you call it, your attention is better spent focusing on the program you are trying to write. Assuming that functions others have written work properly, and not taking the time to re-implement them yourself, is what makes it possible to write truly massive programs containing millions of lines of code.

One final note about the print function: it represents one of the biggest changes between Python version 2 and Python version 3. In Python 3, print is a function—it needs to be called with parentheses. In Python 2, print is a special operator that can be called without parentheses:

```
print "hello, world!"
```

For clarity, the above example is valid only in Python 2, not Python 3. In this textbook, we exclusively use Python 3, but you will often find examples online that still use the Python 2 way of printing. If you translate the Python 2 print commands into the Python 3 print functions, most Python 2 examples also work in Python 3.

That's all we'll say about printing for now. In later chapters, we'll devote significantly more time to more sophisticated forms of output like files, spreadsheets, and web pages. Now, let's turn to input.

4.3 Handling Input

So far, all of the examples we've considered have been simple. Each program has been good for exactly one purpose, printing a single president's name or

converting a single temperature from Celsius to Fahrenheit. The missing component is input—data that the user can give to the program. By taking input, you are able to write a single program that can convert any temperature to Celsius or calculate the amount that a lawyer should bill. In other words, taking input makes a program general, applicable to a wide range of scenarios. By writing a general program, you save yourself the trouble of rewriting it for every new temperature you want to convert to Celsius.

There are two ways of providing input to a Python program: command line arguments and user input.

4.3.1 Command Line Arguments

When you execute a program from the shell, you enter python3 followed by the name of the file that contains the program. If we wanted to run a program called convert_to_celsius.py, we would enter:

$ *python3 convert_to_celsius.py*

Python allows you to encode extra information within this command that is sent to the program as input. These values are written after the name of the program and are separated by spaces. In the example below, the program will receive the string '70' as input, which it can then use as the temperature that it will convert from Fahrenheit to Celsius.

$ *python3 convert_to_celsius.py 70*

The program itself can access this value with a few simple Python commands. Read the code below, and don't worry if part of it are hard to decipher: a description of the mysterious statements will follow.

```
1  import sys
2
3  (1) deg_f = int(sys.argv[1])
4  deg_c = int((deg_f - 32) * 5 / 9)
5
6  print(str(deg_f) + ' fahrenheit')
7  print(str(deg_c) + ' celsius')
```

Line 1 instructs Python that the program will need to use the sys library, which contains a number of useful functions. Among other purposes, this library allows the program to access the command line arguments. You will see import statements from time to time as we use libraries to augment Python with additional functionality.

We access the argument passed to the itself. To do so, write sys.argv followed by the numerical position of the argument that you wish to access placed within square brackets ([ ]). The first command line argument (counting from left to right in the command that the user typed on the shell) is numbered 1. A program could demand multiple arguments, which would be accessed using sys.argv[2], sys.argv[3], etc.

Since Python doesn't know what type of data to expect, command line arguments are always provided to the program as strings. While on line 3, the program converts the string '70' into the integer 70 using the int() typecasting function. Lines 4 through 7 are identical to the code presented earlier; they convert the temperature to Celsius, cut off the decimal by casting the floating point result into an integer, and print the results.

A user can execute this program on any input.

```
$ python3 convert_to_celsius.py 70
70 fahrenheit
21 celsius
$ python3 convert_to_celsius.py 212
212 fahrenheit
100 celsius
$ python3 convert_to_celsius.py 32
32 fahrenheit
0 celsius
$ python3 convert_to_celsius.py -80
-80 fahrenheit
-62 celsius
```

As always, there are plenty of opportunities for errors. Suppose a malicious user decided to derail the program by providing text rather than a number as a command line argument.

```
$ python3 convert_to_celsius.py cat
Traceback (most recent call last):
  File "convert_to_celsius.py", line 3, in <module>
    deg_f = int(sys.argv[1])
ValueError: invalid literal for int() with base 10: 'cat'
```

This error should look familiar; since sys.argv[1] is the string 'cat', it cannot be converted to an integer, so the int function crashes. Now that you are working with full programs, however, Python provides a little bit of extra information in the error messages that we didn't see before. It explains that the error occurred on line 3 and reproduces the offending statement, making it easier to pinpoint the part of the program that caused the error.

You can also trigger a different error if you don't provide as many command line arguments as the program expects.

```
$ python3 convert_to_celsius.py
Traceback (most recent call last):
  File "convert_to_celsius.py", line 3, in <module>
    deg_f = int(sys.argv[1])
IndexError: list index out of range
```

This message is slightly more cryptic. On line 3, Python tried to access argument 1 using the expression sys.argv[1], but the argument didn't exist. Here, the number 1 is known as an index. Since no argument exists at position 1 or beyond, the expression has requested an argument that is "out of range."

4.3.2 Multiple Arguments

Consider a program with more than one command line argument. Suppose that a lawyer wants to calculate the amount to bill a client. She bills a particular amount per hour and knows how many six-minute increments she has worked. The program will take these two items as command line arguments and calculate the overall bill.

```
1  import sys
2
3  price_per_hour = float(sys.argv[1])
4  price_per_six_minutes = price_per_hour / 10
5
6  six_minute_increments = int(sys.argv[2])
7  total_bill  = six_minute_increments * price_per_six_minutes
8
9  print('Total  bill :  $' + str(total_bill))
```

As before, it needs to include the first line of code, which grants the program access to the sys library and the command line arguments. On line 3, the program reads the first command line argument, which is the lawyer's billing rate. Since it is provided as a string, it needs to be converted into a more useful form for calculating. In this case, the program converts it into a float, since it is conceivable that a lawyer would bill at a rate that contains numbers after the decimal point, for example $175.50 an hour. From this value, line 4 can calculate the amount that is billed for each six minute increment by dividing the hourly billing rate by 10.

To access the second command line argument, line 6 uses the expression sys .argv[2], which asks the sys library to provide the argument at position 2. The program converts this argument into an integer, working under the assumption that a lawyer will not bill in fractional six-minute increments. Finally, it multiplies the number of increments by the amount billed for each increment on line 7, arriving at a total bill that is printed on line 9.

A user can now run the program on any inputs she likes.

```
$ python3 bill_calculator.py 250 36
Total  bill :  $900.0
$ python3 bill_calculator.py 175.50 27
Total  bill :  $473.85
```

Helpful tip. If a command line argument needs to include spaces, enclose it in double quotes so that Python recognizes it as a single argument rather than multiple arguments. The quotes will not be passed along to Python.

Style note. Pay careful attention to the way in which the programs in this section have been structured. Just like a piece of writing, programs have been separated into "paragraphs" of logically related code with an empty line in

between. Doing so makes the program easier on the eyes and helps a reader mentally divide the program into smaller components with individual purposes. Also notice that the variables have long names that make it easy to understand what they store and how they are being used.

4.3.3 Making Programs Interactive

Although command line arguments are convenient and we will use them extensively throughout this book, they have several shortcomings that need to be addressed with a different way of accepting input. Command line arguments can be entered only at the beginning of the program, so a program can't decide whether to ask the user for more input on the fly. This also means that the user has to know exactly which input to provide ahead of time; the program can't print helpful prompts or other guidance. In short, command line arguments are static; they don't allow a user and program to interactively exchange information.

Python has a number of other ways of facilitating input. In this section, we will discuss the simplest way of doing so, although we will see many others throughout the course of this book. In addition to the print function, Python includes the input function, which prompts the user to type in text. To use the function, simply write the expression

input()

which will display a cursor on the shell. A user can type in text and press enter or return to submit it. The text that the user typed in is processed into a string. This string becomes the value of the call to the input function when Python evaluates it. So if the user typed in the text

Hello, world!

and then pressed enter, the value of the expression input() would be the string

'Hello, world!'

Notice that calls to the input function are followed by empty parentheses. The function doesn't need any arguments, but the parentheses are still necessary in Python to signify that you are calling the function.

The result of calling the input function is typically saved to a variable for later use. (After all, it would be wasteful to request input and then promptly throw it away.) Below is a simple program that prompts the user for input and echoes it back out to the terminal.

```
1 print('Please type some input below.')
2
3 user_input = input()
4 user_input_length = len(user_input)
5
6 print('The input was ' + user_input)
7 print('It was ' + str(user_input_length) + ' characters long')
```

The first line prints out some text exactly as we did in the section on printing. This text serves as a friendly indication that the user should enter some text in response to the call to the input function that immediately follows. When writing interactive programs, it is good style to clearly prompt the user before asking for input to be sure that he or she knows which information is requested and the format in which it should be provided. Should phone numbers be entered with dashes? Should all four digits of a year be entered with a date?

On line 3, the program calls the input function, which displays a cursor on the shell. This cursor will appear on a line on its own immediately below the text that was just printed. Since the call to input is on the right side of a variable assignment operator (=), the text that the user types is stored to the variable user_input as a string. Line 4 calculates the number of characters in the user's input using the len function we discussed in the previous chapter. Finally, line 6 echoes the text that the user provided, and line 7 prints the length of the input.

Consider a sample execution of the program below. Recall that all text typed by the user is displayed in blue.

```
$ python3 input_basics.py
Please type some input below.
saxophone
The input was saxophone
It was 9 characters long
```

In the interaction above, it is inconvenient that the prompt to the user appeared on a separate line from the cursor where the user typed the input. Python provides a way to put both on the same line by calling the input function with the prompt as an argument. Instead of writing

```
1 print('Please type some input below.')
2 user_input = input()
```

we could instead write

```
1 user_input = input('Please type some input: ')
```

Below is an updated interaction with the modified program.

```
$ python3 input_basics.py
Please type some input: clarinet
The input was clarinet
It was 8 characters long
```

Each of the programs from the previous section can be rewritten using the input function rather than command line arguments.

```
1 deg_f = int(input('Degrees Fahrenheit: '))
2 deg_c = int((deg_f - 32) * 5 / 9)
3
4 print(str(deg_f) + ' fahrenheit')
5 print(str(deg_c) + ' celsius')
```

The only substantive change from the old program appears on line 1, where the program calls the input function rather than accessing a command line argument. The import sys statement was eliminated since the program no longer needs the sys library. Notice that line 1 had to convert the user-provided input from a string, which is what the input function creates, to an integer.

It is perfectly legal to put calls to functions inside calls to other functions. On line 1, Python initially tries to evaluate the call to the int function. To do so, it has to evaluate the function's argument, which is the call to the input function. After the user types in the temperature (say 51), the call to input can be evaluated to the string '51'. Python can evaluate the call to int now that it knows the argument, producing the integer 51.

```
$ python3 convert_to_celsius.py
Degrees Fahrenheit: 51
51 fahrenheit
10 celsius
```

Notice that, in the shell command that tells your computer to execute the program, all of the command line arguments have been removed, leaving only

```
python3 convert_to_celsius.py
```

We can convert the bill calculation program in a similar manner. Since it requires multiple inputs, it will need to make multiple calls to the input function. In this situation, helpful prompts are essential; otherwise, a user will have no idea which number to enter first.

```
1 price_per_hour = float(input('Hourly billing rate: $'))
2 price_per_six_minutes = price_per_hour / 10
3
4 six_minute_increments = int(input('Six minute increments: '))
5 total_bill = six_minute_increments * price_per_six_minutes
6
7 print('Total bill: $' + str(total_bill))
```

As with the temperature conversion program, the import statement was eliminated, and command line arguments have been replaced with calls to the input function. The calls to input have been placed inside type-casting functions (float and int) to ensure that the inputs are converted from strings to forms amenable to calculation. A user can interact with the program by typing each of the requested inputs.

```
$ python3 bill_calculator.py
Hourly billing rate: $150.30
Six minute increments: 182
Total bill: $2735.46
```

Command line arguments vs. the input function. There are tradeoffs between using command line arguments and using the input function; neither

is strictly better than the other. Command line arguments are convenient when a program requires a fixed set of a few small inputs, each of which is known ahead of time. The input function, in contrast, is more flexible and user-friendly, making programs interactive. It is also more conducive to longer input strings, which are cumbersome to type as a single command on the shell. In many cases, however, it is better that a program run without any interaction, a situation where command line arguments are preferable. In general, both forms of input are acceptable, and the choice between the two is a matter of circumstance and taste.

4.4 Text Files

4.4.1 Introducing Text Files

So far, all of the ways of storing data we have discussed have been temporary. As soon as your program ends, all of its variables disappear. When you close your terminal window, the input and output vanish too. Every time your program runs, it starts fresh with no recollection of any of its past executions. The inability to store information across multiple executions of the same program can be a significant limitation in practice. For example, a billing calculator can't keep track of the running total balance over multiple sessions; every time the program starts, the initial balance reverts back to $0.

We can give Python the power to store data in a fashion that will persist across multiple executions of a program (or even across multiple executions of different programs) by using files. A file is a piece of information (like a document or a picture) that is stored on your computer's hard drive rather than in its memory. Memory is the scratch paper that your computer makes available to a program while it is running. A program uses memory to store variables, perform calculations, and compute results. After a program is done executing, Python erases its memory to make it available for other programs to use. (We acknowledge that the term memory is confusing, since your computer does not remember the values a program stores in memory after it terminates.) In contrast, the information on a hard drive is stored permanently, disappearing only when you choose to delete it or the physical hardware dies. In summary, files stored on a hard drive effectively live forever, while variables stored in memory disappear as soon as the program is finished. In previous sections of this chapter, we explored variables; here, we will introduce files.

You are already very familiar with managing files in your day-to-day life. Word-processing documents are files, as are pictures, PDF documents, and the Python programs that you write. Each file you create has a unique name, like `brief.docx`, `vacation.jpg`, `report.pdf`, or `hello_world.py`. Notice that each of these names has some text before and after a dot. The text before the dot is the name of the file. The text after the dot is the file's extension. An extension specifies the kind of information being stored in the file. For example, a file ending in `jpg` stores an image, while a file ending in `py` stores a Python

program. All digital data is made up of 0's and 1's that, when interpreted in a certain way, have meaning that allows your computer to understand them as numbers, strings, pictures, or other kinds of information. Files, as digital data, are also made up of 0's and 1's, and a file's extension tells your computer exactly how to interpret those 0's and 1's in order to show you a PDF document, picture, video, web page, Python program, etc.

In this section, we will focus specifically on text files—files that store a single string. Text files are the simplest variety of files, and they will be a valuable tool throughout the rest of this book. You have already worked with text files in this chapter. Every Python file that you create is a text file storing one long string that happens to contain a well-formed Python program. You can store any string you want in a text file, from the text of Moby Dick to a single letter (or no letters at all). When text files have no other special purpose (like storing a Python program), they are typically given the extension `txt`. For example, a text file storing the running balance of a billing program might be named `balance.txt`. With that said, you can give a text file any extension that you like provided you always treat it as a text file when you try to use it.

4.4.2 Writing to Text Files

Saving and restoring data from a text file is slightly more involved than creating or using a variable. First, you need to open a text file, which gives your Python program the ability to manipulate it. This process is similar to opening a document in a word processor before trying to read or edit it. You can open a file with the aptly named open function. This function takes two arguments: the string name of the file and a string signifying whether you intend to read from the file or write to the file. If you are reading, the second argument should be the string `'r'`; if you are writing, it should be `'w'`.

When you call the open function, Python will look for the specified file in the same location on your computer (that is, the same folder or directory) as that in which your Python program resides. (If you open a file in interactive Python, it will open in a special directory where interactive Python is stored.) In later chapters, we will discuss the concept of directories in more detail and learn how to manipulate files in other locations on your computer. For now, the files to which your program reads and writes must be saved in the same location on your computer as the program itself.

The open function evaluates to a special Python type called a filehandle. When Python opens a file, it establishes a line of communication between your Python program and your computer's operating system, which manages the files stored on your computer. A filehandle is the means by which you can send your operating system commands for manipulating a particular file.

```
>>> fh = open('preamble.txt', 'w')
```

In the statement above, we have opened a file called `preamble.txt` in write-mode (signified by the string `'w'` as the second argument). The call to the open function evaluates to a filehandle, which we store in the variable fh.

Opening a file in write-mode makes it possible to save information to the file. However, be careful—opening a file in write-mode wipes out the file if it already exists and creates a new, empty file with the specified name. If the file doesn't yet exist, opening it in write-mode will create it. This behavior is similar to saving a value to a variable: the old value of the variable, if any, is wiped out and the variable is created anew if it doesn't already exist.

A word of warning. Be very careful that you don't try to open an important file in write-mode—doing so deletes the existing version of the file, and there is no way to undo the operation.

Back to our example. Now that we have a communication channel with the file in the form of the filehandle, we can operate on the file. At this juncture, we will discuss only the most basic reading and writing operations. We will continue to introduce new functions for manipulating files as you gain more experience with Python. As the name suggests, when a file is open in write-mode, you can write to it.

>>> fh.write('We the People')

To write to a file, type the name of the filehandle followed by a dot. After the dot, call write with the string you wish to write to the file in parentheses. The statement above causes Python to save the string 'We the People' inside the file preamble.txt. Although the example above passes a string directly to write, you could use a string of any kind, including a variable that stores a string.

Notice that we called write in a slightly different way than we have called other functions like print and input. First, we typed the name of a variable followed by a dot and then called write with an argument. This dotted notation is subtly different from a normal function call in ways we will explore in later chapters. Although write looks and behaves like a function, it is actually known as a method. The filehandle, fh, is known as the method call's object; for now, you can think of it as another argument to write that is provided using a slightly different notation. This working knowledge is all you need for the moment; we will revisit these concepts in detail in Chapter 7.

Once a file is open in write-mode, you can make repeated calls to write as many times as you like. Each subsequent call to write concatenates the string argument onto the end of the file's current contents. After the call to write above, the file stores the string `'We the People'`. We can then call write again.

>>> fh.write(' of the United States')

This string is added to the end of the file, meaning the file now stores the string 'We the People of the United States'. If we wanted, we could keep calling write to continue adding to the file. When you are done writing, the last step is to close the file, severing the connection between your Python program and the file.

```
>>> fh.close()
```

Helpful tip. If you want to write to a file without wiping out its contents, you can open the file in append mode with the string `'a'` in place of the string `'w'`. Append-mode is identical to write-mode, except that it preserves the existing contents of the file and writes new information to the end of the file. If a file that doesn't yet exist is opened in append-mode, Python will create a new, empty file and then open it.

4.4.3 Reading from Text Files

Once you have created a text file, you can read back its contents. Reading from a file looks quite similar to writing. First, open the file and save the resulting filehandle to a variable. Since we are reading, the second argument to the call to open should be the string `'r'`, signifying that the file will be opened in read-mode.

```
>>> fh_for_reading = open('preamble.txt', 'r')
```

Opening a file in read-mode has no impact on the existing file; it does not modify the file in any way. It simply evaluates to a filehandle that is prepared to read back the string stored in the file.

Since opening files in read-mode is so common, you can leave off the second argument to the open function and Python will automatically assume you intend to open the file in read-mode. Concretely, we could rewrite the statement above as:

```
>>> fh_for_reading = open('preamble.txt')
```

and Python will still open the file in read-mode.

Unlike write-mode, if you try to open a nonexistent file in read-mode, Python will produce an error.

```
>>> fh_nonexistent = open('does_not_exist.txt')
```

In response to this command, Python will display the following error message:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'does_not_exist.txt'
```

Thankfully, this error message is quite self-explanatory—the file does_not_exist .txt wasn't found, so Python could not open the file in read-mode.

Once a file has been successfully opened in read-mode, you can access the string stored in the file using the read method, which takes no arguments and evaluates to a string containing the entire text of the file.

```
>>> preamble = fh_for_reading.read()
>>> preamble
'We the people of the United States'
```

Above, we called read after the name of the filehandle and a dot. It evaluated to the string contents of the file, which we stored to the variable preamble. Finally, we inspected the value stored in the variable preamble, confirming that it was the string we wrote the the file earlier.

Often, you will want to quickly retrieve the contents of a file. You can combine the steps mentioned above into a single line of code. Rather than saving the filehandle to a variable, you can immediately call the read method on the result of calling the open function.

```
>>> preamble = open('preamble.txt').read()
```

First, the call to open evaluates to a read-mode filehandle to preamble.txt. Python then calls the read method on this filehandle, evaluating to the string contents of the file. This string is saved to the variable preamble.

Notice that, since we don't save the filehandle, we don't close the file. After writing to a file, closing is essential; if you do not close the file, some of the information you have written may not be saved properly. After reading from a file, closing is considered good style but generally isn't essential.

4.4.4 Remembering the Balance

We will close this introduction to files by augmenting the billing calculator program to keep track of the running total to be billed across program executions. We will do so by storing the total balance in a text file. Doing so requires proceeding in two steps. First, we will need to write a short program that creates the file and initializes the balance to 0. Once the file has been created, the billing program can read the contents of the file at the beginning, compute the new bill, and write the updated balance back to the file at the end.

We will start by composing a Python program called setup.py. This program's sole job is to create a file called balance.txt and write the value '0'. Once that file has been created, our billing calculator can read in the previous balance from this file, compute the new balance, and rewrite it to the file. We can fit setup.py into three lines.

```
1 fh = open('balance.txt', 'w')
2 fh.write('0')
3 fh.close()
```

On line 1, the program opens the file balance.txt in write-mode by calling the open function with the arguments 'balance.txt' and 'w'. It saves the resulting filehandle to a variable. Line 2 writes the string '0' (representing an inital balance of $0) to the file using the write method. Line 3 closes the filehandle. After running this program once, the file balance.txt will have been created with an initial value of 0. The program bill_calculator .py can now read from and update this file.

Be careful not to run setup.py after you start recording bills—doing so will wipe out the existing balance and reset it to 0.

We will now update bill_calculator.py to use the file balance.txt. As before, the program will receive two command-line arguments representing the hourly billing rate and the number of six-minute increments worked.

```
1  import sys
2
3  price_per_hour = float(sys.argv[1])
4  price_per_six_minutes = price_per_hour / 10
5
6  six_minute_increments = int(sys.argv[2])
7  total_bill  = six_minute_increments * price_per_six_minutes
```

It uses this information to compute the amount billed per six minute increment (line 4) and the total bill (line 7). Now, we will retrieve the existing balance.

```
9  fh_old_balance = open('balance.txt')
10 old_balance = float(fh_old_balance.read())
11 fh_old_balance.close()
```

On line 9, the program opens the file balance.txt in read-mode and saves the filehandle to the variable fh_old_balance. It reads the old balance from this file into the variable old_balance on line 10. Notice that the result of calling read was casted to a floating-point number using the float function. The read method always returns a string, so we need to convert that string into a floating-point number to use it for calculations. Finally, on line 11, we close the filehandle.

Now that we have the old balance, we can calculate the new balance.

```
13 new_balance = old_balance + total_bill
14
15 print('Old balance: $' + str(old_balance))
16 print('New bill: $' + str(total_bill))
17 print('New balance: $' + str(new_balance))
```

On line 13, we add the new amount to the old balance, computing the new total balance. We print these three numbers on lines 15-17. If the old balance stored in balance.txt was $100 and the new bill was $50, the program would print:

```
Old balance: $100
New bill: $50
New balance: $150
```

Finally, we need to write the new balance back to balance.txt.

```
19 fh_new_balance = open('balance.txt', 'w')
20 fh_new_balance.write(str(new_balance))
21 fh_new_balance.close()
```

On line 19, we open balance.txt in write-mode and save the resulting filehandle to the variable fh_new_balance. Recall that opening a file in write-mode deletes the existing version of the file, meaning that the statement on line 19 wipes out the old version of balance.txt and the string that it stores. On line 20, we use

the write method to write the new balance to the file. Since the file stores a string, we have to cast the floating point variable new_balance into a string using the str function before passing it to the write method. Now that the file has been updated with the new balance, we can close the file (line 21).

Consider an example series of executions for these two programs. First, we need to create and initialize balance.txt using setup.py.

```
$ python3 setup.py
```

Afterwards, we can begin using bill_calculator.py to calculate new billing amounts and update the balance accordingly.

```
$ python3 bill_calculator.py 100 3
Old balance: $0.0
New bill: $30.0
New balance: $30.0
```

Since there were no previous bills, the old balance is $0. The new bill is $30, and the total balance is updated to $30.

```
$ python3 bill_calculator.py 150 5
Old balance: $30.0
New bill: $75.0
New balance: $105.0
```

When we run bill_calculator.py a second time, it retrieves the old balance ($30), computes the new bill ($75), and generates the new balance ($105), which it writes back to the file balance.txt.

In summary, files give us a way to save and restore information that survives beyond the execution of any one program, giving our programs the ability to remeber values across executions.

4.5 Comments and Style

Throughout this introduction to Python, we have placed enormous emphasis on style. Programs should be written with a reader in mind, whether that reader is a peer or the same person who wrote the program coming back six months later. Programs that are clear and comprehensible make debugging easier and are far more amenable to changes and upgrades down the road. Lawyers, who experience arcane stylistic requirements from all sides, will understand the need for uniformity and predictability.

Although we have covered only a small fragment of the Python language so far, we have already delivered a number of lessons and mandates about style.

- An extra space should be left around all mathematical operators to make expressions more legible.
- Variables should have descriptive names written in underscore case.

- Programs should be separated into "paragraphs" of related code with empty lines in between.
- Requests for user input should have clear prompts that explain the kind of input that is desired.

These lessons will continue to accumulate throughout the remainder of this book. So far, however, we have yet to thoroughly cover one of the most important tools for maintaining good style—comments. As was previewed in Chapter 2, Comments are annotations that you can add to your Python code, explaining the purpose of a particular expression or summarizing the behavior of a code paragraph. Python ignores comments when it executes a program, but readers will find them to be essential for making sense of your code.

To add a comment into a Python program, simply type a # symbol. Everything you write from the # symbol to the end of the line will be ignored by Python.

```
1 # This is a comment.
2 # This is another comment.
3 print('This is not a comment.') # But this is.
4 x = 10 # Set x equal to 10.
5 x += 1 # Add 1 to x.
6 print(x) # Print x.
7 # This is yet another comment.
8
9 # After a blank line, one more comment.
```

In the program above, Python will ignore everything except lines 3 through 6. And even on lines 3 through 6, Python will ignore the comments that come after the statements. In other words, the output of the program will be:

```
This is not a comment.
11
```

In general, it is good style to ensure that no line of your program is too long. A typical rule of thumb is that you should avoid lines of code longer than 80 (or alternatively 100) characters. Long lines can make your program hard to read, since they are likely to be too lengthy to fit on some people's screens. In addition, a long line is usually an indicator that an expression is too complicated; instead, it should be broken up into several subexpressions, each of which is saved to a variable so that a reader can examine the computation step by step. If you have a comment that would otherwise be too long to fit on one line of code, you can break it up into consecutive comments on multiple lines.

```
1 # This is a comment that is too long to fit on one
2 # line of code. Instead, it is broken up into a
3 # comment that stretches across multiple lines.
```

Here are a few other best practices for using comments effectively.

- Leave one space after the # sign at the beginning of a comment.
- Comments should generally begin with a capital letter and end with a period. That way, someone can easily tell if the end of a comment has accidentally been deleted. If it has, it probably took some of your code with it.
- Each "paragraph" of code should begin with a comment, contain at most a few lines of code, and end with an empty line.
- Comments that occur on the same line as code should be used sparingly, since they can often be distracting.
- Err on the side of adding too many comments rather than too few, but be careful about going overboard. Imagine the information that a reader would need to know and present it concisely. A reader who is overwhelmed with comments will struggle just as much as a reader who doesn't have any at all.

As a case study, consider the temperature conversion program from earlier in the chapter. Comments have been added to the command line argument variant.

```
1  # convert_to_celsius.py: This program converts its first
2  # command line argument, which is the temperature in
3  # Fahrenheit, into the temperature in Celsius.
4
5  import sys # For processing command line arguments.
6
7  # Read the temperature in Fahrenheit and convert to Celsius.
8  deg_f = int(sys.argv[1])
9  deg_c = int((deg_f - 32) * 5 / 9)
10
11 # Print the temperature in both units.
12 print(str(deg_f) + ' fahrenheit')
13 print(str(deg_c) + ' celsius')
```

A longer comment at the beginning of the program introduces its purpose and explains how to use it. It notes that the program expects one command line argument and describes what that argument represents. After reading this comment, a user should be able to properly execute it.

After an empty line, line 5 includes a short comment, which justifies why the sys library is necessary for this program. After another blank line, the first paragraph of code begins with a comment summarizing the tasks it accomplishes. The paragraph ends with a blank line. The final paragraph of code is structured similarly.

Although it may seem tedious initially, you should strive to comment every program you write to the standard set by the example above. You will thank yourself when you begin writing programs that are hundreds or thousands of

lines long. When working with a large program, it is impossible to keep everything in your head at once, let alone over multiple days interspersed with other work. The operating system that your computer is running consists of tens of millions of lines of code, far more than any one person could hope to keep track of. Without thorough comments to document each component, the thousands of software engineers who built it would have had an infinitely harder job. For the comparatively smaller programs that you will write, the same principle applies; comments and good style will save you time and spare you frustration.

Early on, most of the programs you write will be very small, comprising at most tens of lines. These programs will generally be self-explanatory; comments won't be essential for making sense of how they work. It is important, however, that you get into good habits early on so that you are prepared to write and maintain the larger programs that will come a bit later in the book.

4.6 Cheatsheet

Variables

- Variables store values so that they can be reused in other expressions.
- A variable is created and assigned a value using the = operator. The statement x = 5 stores the value 5 in the variable x. You can write any expression to the right of the = operator; Python will evaluate that expression and then store the resulting value in the variable.
- If you assign a value to a variable that already exists, the variable's current value gets overwritten with the new value.
- When you use a variable in an expression, Python replaces all appearances of that variable with the value that it stores. When x stores the value 5, the expression 2 + x evaluates to 7.
- The operators +=, −=, ∗=, etc. update the value of a variable. For example, x += 3 is equivalent to x = x + 3, which adds 3 to the current value of x.
- Variable names are composed of letters, numbers, and underscores. Variable names may not start with a number. It is good style for variable names to take the form of lowercase words separated by underscores. For example, miles_per_hour.

Functions

- A function is a bundle of Python code located somewhere outside of your program. You can invoke this code to perform special tasks like receiving input and producing output.
- You can cause a function to execute by calling it. To call a function, write the name of the function followed by a pair of parentheses. For example, the statement input() calls the input function.
- Some functions require you to provide them with extra information. For example, the print function needs to know which string to print. Each piece of extra information is known as an argument. To pass an argument to a function, place it between the parentheses of the function call. For example, print('hello') calls the print function with the argument 'hello'. When a function requires multiple arguments, they are placed in the parentheses of the function call and separated by commas.
- Functions we have studied previously include type-casting functions (int(), float(), and str()) and the string length function (len()). Each of these functions takes one argument.

- Some functions evaluate to a value that you must save to a variable. For example, input, len, and the type casting functions. Other functions (like print) make a change to the outside world (like printing to the shell) but do not evaluate to a value that must be saved.

Printing

- Printing causes Python to display output to the shell.
- To print a string, call the print function with the string as its argument. For example, the statement `print('hello')` outputs the string `'hello'` to the shell.

Command-line Arguments

- A command-line argument is a form of input that is provided to the program before it begins executing.
- To pass command-line arguments to a program, type after the name of the program being executed. If you are supplying more than one command-line argument, the arguments are separated with spaces. For example,

  $ *python3 my_program.py alice 21 hello*

  executes the program my_program.py with three command-line arguments (alice, 21, and hello).
- To access the values of the command-line arguments:
  - Add the line `import sys` at the beginning of your program.
  - Write the expression `sys.argv[1]` to access the first command line argument (going left to right), `sys.argv[2]` to access the second, etc.
- All command-line arguments are treated as strings. If you need them in the form of a different type, you will need to perform the appropriate conversions.

Input

- The input function allows your program to request input from the user as it is executing.
- Calling the input function causes a cursor to appear on the shell. A user can type in text. When the user presses enter, the text is converted into a string. The call to the input function evaluates to this string.
- You should store the result of calling the input function to a variable.
- When you call the input function without an argument, it will display a cursor to a user on its own line. When you call the function with a string argument, it will display the value of that argument before the cursor.

Text Files

A file is an entity that is stored permanently on your computer's hard drive. Examples of files include documents (brief.docx), pictures (vacation.jpg), and Python programs (bill_calculator.py). A text file is a specific kind of file that stores a single string. Each file has a name (the part before the dot) and an extension (the part after the dot). Text files typically have the extension txt.

To write to a file:

1. Open the file using the open function. The function takes two arguments: a string filename and a string mode. When writing to a file, the second argumeng should be 'w', signifying write-mode. This function returns a filehandle, which should be stored to a variable.

   >>> fh = open('presidents.txt', 'w')

   Opening a file in write-mode creates the file anew and empty, deleting an old version of the file if it already exists.

2. To write to the file, use the write method on the filehandle variable and pass it the string you wish to write.

   >>> fh.write('Washington')

3. You can call write repeatedly. Doing so will concatenate its argument onto the end of the current contents of the file.

   >>> fh.write('; Adams')
   >>> fh.write('; Jefferson')

4. When you are done, be sure to close the filehandle.

   >>> fh.close()

To read from a file:

1. Open the file using the open function. The second argument should be `'r'`, signifying read-mode. You can leave off the second argument completely and Python will still open the file in read-mode.

   >>> fh = open('presidents.txt')

2. To read the entire contents of the file into a string, use the read method on the filehandle variable. It will evaluate to a string, which you can save to a variable.

   >>> presidents = fh.read()
   >>> presidents
   'Washington; Adams; Jefferson'

3. When you are done, be sure to close the filehandle.

   >>> fh.close()

Appending to an existing file. To write to the end of an existing file, open the file in append-mode by passing open the string `'a'` as its second argument. Doing so will open the file for writing without deleting the existing version of the file. Any calls to write will concatenate onto the end of the file.

Comments

- A comment is a line of code that Python ignores. Comments are valuable for documenting your program for future readers (and your future self).
- Begin a comment by typing the # character. Python ignores verything from the # character until the end of the line.
- It is good style to leave a space before the # character, begin the first word of the comment with an uppercase letter, and end the comment with a period.
- It is good style to break your code into short "paragraphs," each of which begins with a comment, contains a handful of lines of code, and ends with a blank line.

Keywords

Argument—A piece of information passed to a function when it is called. Example: the string passed to the print function is its argument. In the statement `print('hello')`, the argument to the print function is the string `'hello'`.

Assign—To store a value in a variable. Example: the statement x = 5 assigns the value 5 to the variable x.

Call—To cause a function to execute. In Python, this takes the form of typing the function's name followed by opening and closing parentheses. Any arguments to the function are included between the parentheses. Example: the statement `print('hello')` calls the print function.

Copying By Value—When assigning the value of one variable to another (i.e., x = y), the value of the variable on the right is copied into the storage space for the variable on the left. Afterwards, changing one variable doesn't cause the other to change. All of Python's basic types (integers, floating point numbers, strings, booleans, and none) are copied by value. Other types we will encounter later are not copied by value.

Define—To cause a variable to come into existence. The first time a variable is assigned to, it is also defined.

Extension—The part of the filename that comes after the dot. It determines how your computer interpretes the contents of the file. Examples: `jpg`, `pdf`, `py`, `txt`.

File—Information stored permanently on your computer's hard drive. Example: word-processing documents, pictures, PDFs, and Python programs.

Filehandle—A special Python type representing a communication channel between your Python program and a particular file on your computer. By operating on this filehandle, you can manipulate a file.

Function—A bundle of code elsewhere in a program that takes command-line arguments as input and evaluates to an output value (i.e., len()) and/or has some effect on the outside world (i.e., print()). When a function is called, Python temporarily stops executing your program, executes the function's code, and then returns to your program when the function is done.

Hard drive—Your computer's long-term storage area. Data stored to your hard drive remains there indefinitely until you explicitly delete it or the physical hardware breaks. A computer's memory and its hard drive are two distinct storage areas.

Object—The value written before the dot in a method call. See: method.

Memory—The space your computer provides to each running program for keeping track of its variables. As soon as a program is finished executing, its memory is erased and recycled for other programs to use. A computer's memory and its hard drive are two distinct storage areas.

Method—A function that is called using slightly different syntax. The funtion call is written after a value (like the name of a variable) and a dot. The value is known as an object. Method calls behave in a manner identical to function calls, and the object is just another argument. Example: given a filehandle opened in read-mode and stored in the variable `fh`, you can read the contents of a file using the read method on the filehandle object: `fh.read()`.

Reassign—To overwrite the value of an existing variable with a new value.

Reserved Word—A name that cannot be used as the name of a variable because it already has another meaning in Python. Example: True and False are not valid variable names because they are boolean values.

Standard Library—The set of functions that come built-in to Python.

Statement—Expressions that do not evaluate to a value (or equivalently that always evaluate to None). Example: any call to the print function is a statemnet, since the print function does not evaluate to a value.

Text file—A file that stores a single string. Example: a `py` file, which stores a very long string that happens to be a well-formed Python program.

Variable—A name and associated storage space that makes it possible to store a Python value for later reuse.

Chapter 5

Conditionals

The Python programs you have written so far suffer from one key limitation: they always perform the same operations in the same order every time they run. In many cases this behavior is acceptable—perhaps even encouraged. It would be disconcerting, for example, if a program used a different sequence of steps each time it converted a temperature from Celsius into Fahrenheit. (In this case, it would probably be a sign that the program is wrong.)

However, few programs are this simple. Most useful programs will need to change their behavior depending on the inputs they are given. For example, your banking website should grant you access if you type in the right password but display an angry error message if you don't. And if you mistype that password enough times, it should lock you out completely for a few minutes. The Python you know so far is insufficient for building a system that needs to adapt its behavior in this manner.

Consider something even simpler—a program that divides two integers.

```
1  import sys
2
3  # Take two integers as command-line arguments.
4  num1 = int(sys.argv[1])
5  num2 = int(sys.argv[2])
6
7  # Divide the numbers and print the output.
8  result = num1 / num2
9  print(str(result))
```

In most cases, this program will work just fine.

```
$ python3 divide.py 5 2
2.5
```

But, since division by zero is (mathematically speaking) illegal, this program will crash if a user tries to divide by zero.

```
$ python3 divide.py 8 0
```

```
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    result = num1 / num2
ZeroDivisionError: division by zero
```

Ideally, the program shouldn't crash, even when a user provides this admittedly nonsensical input. Instead, it would be much more user-friendly to detect that a user has made this mistake, print a helpful reminder that division by zero isn't allowed in mathematics, and end the program without trying to perform any ill-fated division. This approach would avoid inelegant crashes and better assist a user who might otherwise be puzzled by Python's cryptic error messages. But to make this happen, the program would need to do something special when a user inputs zero as the second command-line argument—it would need to change its behavior on the fly. Python (and any other programming language) provides a way to do so using an if-statement.

```
1  import sys
2
3  # Take two integers as input.
4  num1 = int(sys.argv[1])
5  num2 = int(sys.argv[2])
6
7  # If num2 is zero, print a helpful reminder.
8  if num2 == 0:
9      print('Division by zero is illegal!')
10
11 # Otherwise, divide the numbers and print the output.
12 else:
13     result = num1 / num2
14     print(str(result))
```

The program can now gracefully handle the input that crashed it before.

```
$ python3 divide.py 8 0
Division by zero is illegal!
```

If-statements, or conditionals as they are known formally, give programs the power to adapt depending on inputs, the day of the week, or any other factor that might vary from execution to execution. They are a central element of any useful program, so we will discuss their intricacies at length in this chapter. We will do so by way of the tortuous conditional outcomes of Medicare, which provides health insurance to elderly and disabled Americans. Tens of millions of people depend on its benefits, but the nuances of eligibility rules and registration windows for various components of the program are nearly impossible for a busy lawyer to remember (let alone a busy layperson). Instead, we can formalize this information in a computer program, providing advice about Medicare that is accessible to users without the need for human assistance.

5.1 Booleans Revisited

Boolean expressions are the essential ingredient that make conditionals possible. A conditional instructs Python to execute one or more lines of code if a specific boolean expression evaluates to True; otherwise, Python should skip those lines of code. In other words, when a conditional asks whether to execute certain lines of code, it phrases its question in the form of a boolean expression. It is therefore impossible to understand conditionals without a thorough understanding of booleans. (It is worth rereading Section 3.4 before continuing further.) Here, we briefly review boolean expressions.

Boolean basics. Booleans are one of Python's five fundamental types. There are only two possible boolean values: True and False. Boolean expressions deduce whether a particular condition does or does not hold. For example, given an integer variable x, the boolean expression x > 3 will evaluate to True when x is bigger than 3 and will evaluate to False when x is smaller than or equal to 3. One way to understand boolean expressions—the perspective we take in this chapter and going forward—is that they ask a question. The aforementioned expression asks, "is the variable stored in x greater than the integer 3?" From this point of view, the boolean value True means the answer is "yes;" the boolean value False means the answer is "no."

Python offers a rich vocabulary through which to ask such questions. The two most important operators are == (equality) and != (disequality). The == operator asks whether two values are equal to each other. For example, the expression x == y + 3 asks, "does the value of the expression x equal the value of the expression y + 3?" Once again, this expression will evaluate to True if the answer to the question is "yes" and False otherwise.

The != operator asks the opposite question. The expression x != y + 3 asks, "are the values of the expressions x and y + 3 unequal?" or, converted into more comprehensible language,"are the values of the expressions x and y + 3 different?" If they are indeed different, Python answers "yes" by evaluating the expression to the boolean value True; if they are the same (that is, they are equal to each other), Python answers False.

In Chapter 3, we discussed a variety of different operators that ask questions and produce boolean values. For example, the operator > asks whether the value on the left is bigger than the value on the right. There are similar <, >=, and <= operators. For a complete list, consult Section 3.4 or the Chapter 3 cheatsheet.

Two final notes:

1. Although the examples above all used integers, these operators are equally valid on all of the fundamental types (floating point numbers, strings, booleans, and even the none type). One could ask, "does the variable password equal the string '12345'?" with the expression password == '12345'. Likewise, one could ask whether the variable temperature is bigger than 98.6 with the expression temperature > 98.6.

2. Python allows you to put entire expressions to the left and right of these operators. Python will evaluate the expressions before comparing them. For example, the expression 5 * x > 2 * y + z asks the question, "is (5 times x) bigger than (two times y plus z)?"

Compound operators. The aforementioned operators are useful for asking individual questions. "Is x bigger than y?" "Does justice equal 'Ginsburg'?" This machinery isn't sufficient for asking compound questions. "Is the variable lsat at least 120 and no bigger than 180?" "Is the variable act greater than 30 or is the variable sat greater than 1400?"

Python offers two tools for asking these kinds of questions. The first is the and operator. Suppose we wanted to ask, "is the variable lsat between 120 and 180?" This question is really two different questions: "Is the variable lsat greater than or equal to 120 and, separately, is it less than or equal to 180?" The answer to this question is only "yes" if the answers to both separate questions are "yes." Consider some examples.

- Is an LSAT score of 150 greater than or equal to 120? Yes. Is it less than or equal to 180? Yes. Finally, we have enough information to conclude that, yes, the score is between 120 and 180.
- Is an LSAT score of 190 greater than or equal to 120? Yes. Is it less than or equal to 180? No. Since the answer to the second question is no, it can't be between 120 and 180.
- Is an LSAT score of 12 greater than or equal to 120? No. Is it less than or equal to 180? Yes. Since the answer to the first question is no, it can't be between 120 and 180.

In cases where we want to ask whether the answers to two separate questions are both "yes," they can be combined together with the and operator. To ask our question about LSAT scores, we would use the boolean expression lsat >= 120 to ask whether the LSAT score is bigger than 120 and the boolean expression lsat <= 180 to ask whether the LSAT score is smaller than 180. To ask whether the answers to both questions are simultaneously "yes," we would write lsat >= 120 and lsat <= 180. This expression only evaluates to True when the expressions on the left and write each evaluate to True.

Python has a similar operator that asks whether the answer to at least one of the questions is "yes." For example, suppose we wanted to assess eligibility for an internship by asking whether a user either had an ACT score greater than 30 or an SAT score greater than 1400. This is also two separate questions. "Is the ACT score greater than 30?" "Is the SAT score greater than 1400?" If the answer to either one of these questions is "yes," the answer to the overall question is "yes." Python provides the or operator for cases where the answers to at least one of two questions must be "yes." We can ask the question about the ACT score with the boolean expression act > 30. We can ask the question about the SAT score with the boolean expression sat > 1400. We can ask the

compound question with the expression act > 30 or sat > 1400. If the answer to at least one of these questions is True, then the overall expression will evaluate to True.

Python offers one final compound operator: not. The not operator flips the value of a boolean expression from True to False and vice versa. For example, if the boolean expression x > 5 evaluates to False, the expression not(x > 5) evaluates to True. The not operator can be useful from time to time, so it's important to keep in in mind. Notice that, using the not operator, there are usually several different ways to write the same boolean expression. For example, the expression y + 3 != 12 is equivalent to not(y + 3 == 12). Use your best judgment to decide on the variant that is most readable in any particular situation.

5.2 If-Statements

Now that we have completed a refresher on boolean expressions, we are ready to put that knowledge to use in conditionals. Over the coming pages, our goal will be to write a series of programs that help a user understand whether he is eligible for Medicare, when he should register for the program, and when to elect various benefits that the program provides. This program will take the form of a back-and-forth set of questions from the program and answers from the user. The program will need to change its questions on the fly depending on the user's answers. For example, if the user is over 65, then registration penalties depend on whether she is still working full-time and receiving insurance from an employer.

We will begin by writing a program that answers the most basic question about Medicare: is the user eligible? The answer is simple—if the user is at least 65 years old, he is eligible. We can encode this question in Python with the code below. It uses an input statement to directly ask the user his age.

```
1 # Ask for the user's age.
2 age = int(input('How many years old are you? '))
```

The user is expected to answer with their age (written as a whole number). We embed this assumption into our code by passing the result of the call to the input function into the int function. Recall that this converts the string produced by input into an integer that we can use for mathematical purposes.

At this point, our program should either respond that the user is eligible (if the user enters an age that is greater than or equal to 65) or that age alone does not qualify the user for Medicare (if the user enters an age that is less than 65). Our program will need to somehow print one of two different messages depending on the input, a feat we could not accomplish until this chapter.

We will achieve this behavior with an if-statement. An if-statement allows a program to execute code conditionally—only in particular circumstances. In our case, we want to print a specific message only in the case where the user answers the question with a number that is at least 65.

```
4  # If the age is at least 65, then the user is eligible.
5  if age >= 65:
```

If-statements consist of four parts.

1. Type the command `if`, which tells Python you are beginning an if-statement. The word `if` is known as a keyword in Python—it is reserved exclusively for use in if-statements, so Python will complain if you try to give that name to a variable.

2. After leaving a blank space, type a boolean expression. This boolean expression is the question that the if-statement tests. (It is often referred to as the condition, test, or guard of the if-statement.) If the answer to the question is "yes," then the code guarded by the if-statement executes; if the answer is "no," then Python skips the code completely. Equivalently, f the boolean expression evaluates to True, then the code guarded by the if-statement executes; if not, then Python skips it completely. This is how Python manages to execute code conditionally.

   In our case, we want to check whether the integer with which the user answered our question was at least 65, so we will use the following boolean expression in our if-statement:

   ```
   age >= 65
   ```

   You will recall that this expression evaluates to True only when the value of the variable age is greater than or equal to 65 and False otherwise.

3. Type a colon (`:`) at the end of the line of code that began with `if`. This colon tells Python that you've finished the boolean expression.

4. Starting with the following line of the program, type one or more lines of code that Python should run if the boolean expression evaluates to True. Each of these lines needs to be indented with blank spaces. This indentation tells Python that the code is part of the if-statement, separate from the main program. Collectively, these lines are known as the body of the if-statement.

If the boolean condition evaluates to True, then the indented lines of code are executed; if it evaluates to False, then the indented lines are skipped. Either way, any unindented lines that follow the if-statement run afterward, and the normal flow of Python execution resumes.

```
4  # If the age is at least 65, then the user is eligible.
5  if age >= 65:
6      # This is some indented code within the if-statement.
7      # It is only executed if the boolean condition above
8      # was True.
9      print('You are eligible for Medicare!')
10     print('Make sure you are registered!')
```

```
11
12 # This code isn't indented, so it is executed regardless
13 # of whether the boolean condition in the if-statement
14 # above was True.
15 print('Have a nice day!')
```

In this case, lines 6 through 10 were indented, so they will run whenever the user types in an integer larger than 64. Lines 12 through 15 will execute no matter what, since they aren't indented.

```
How many years old are you? 67
You are eligible for Medicare!
Make sure you are registered!
Have a nice day!
```

If the user types an age less than 65, then lines 6 through 10 will be skipped.

```
How many years old are you? 42
Have a nice day!
```

There are a few important rules you should know about indentation. When you indent many consecutive lines of code within an if-statement (known collectively as a code block), they all need to be indented by the same amount. You can indent with as many tabs or spaces as you like, but the indentation needs to be identical among every line of code in the block. It is considered good style to indent with four spaces. (It is considered bad style to ever use tabs.) If some lines are indented more than others—that is, if the lines of a code block are not indented evenly—you will see an error.

```
1 age = int(input('How many years old are you? '))
2
3 if age >= 65:
4     # The line below this one is indented unevenly!!!
5       print('You are eligible for Medicare!')
6     print('Make sure you are registered!')
```

When you try to execute the above program, it complains about the uneven indentation.

```
  File "medicareBot.py", line 5
    print('You are eligible for Medicare!')
                                ^
IndentationError: unindent does not match any outer indentation level
```

This error is Python's way of saying that every line of a code block should be indented to the same level.

5.3 Multiple If-Statements

Your code can contain as many if-statements as you like. Python will evaluate each if-statement in order as it moves through a program from top to bottom.

```
1  # Ask for the user's age.
2  age = int(input('How many years old are you? '))
3
4  # If the user is older than 65, he is eligible .
5  if age >= 65:
6      print('You are eligible for Medicare!')
7
8  print('No more legal questions, but ... ')
9
10 # Check if it's raining.
11 raining = input('Is it raining? ')
12 if raining == 'y':
13     print('Oh no! I forgot my umbrella today!')
14
15 print('See you later!')
```

No matter how the user responds to the question about age on line 2, Python will resolve the if-statement on lines 5 and 6 and resume normal execution of the program on line 8. After asking the weather on line 11, Python will then encounter the second if-statement on line 12. Just as before, it will execute the code block inside (line 13) if the boolean guard raining == 'y' is True and skip it otherwise. Either way, the program will conclude with the print statement on line 15.

```
How many years old are you? 37
No more legal questions, but ...
Is it raining? y
Oh no! I forgot my umbrella today!
See you later!
```

Suppose instead that the user is 70 but the weather is sunny.

```
How many years old are you? 70
You are eligible for Medicare!
No more legal questions, but ...
Is it raining? n
See you later!
```

This time, the boolean condition of the if-statement on line 5 will be True, causing it to print the message on line 6. Line 8 is no longer indented, so Python will resume normal execution, running this print statement and the input statement on line 11 no matter what. Finally, the boolean condition of the if-statement on line 12 will be False, causing Python to skip the indented code on line 13 and proceed to the final print statement on line 15.

5.4 Variables and If-Statements

The code inside an if-statement can request input; print, create or modify variables; or do anything else you can do in a normal Python program. For example, you could store the goodbye-message that the program prints as a variable and change it depending on how the program proceeds.

```
1  # The default goodbye message.
2  goodbye = 'Sorry that you are not eligible :( '
3
4  # Ask for the user's age.
5  age = int(input('How many years old are you? '))
6
7  # If the user is older than 65, he is eligible .
8  if age >= 65:
9      print('You are eligible for Medicare!')
10     goodbye = 'Congrats on the good news!'
11
12 print('See you later! ' + goodbye)
```

If the program doesn't enter the if-statement, then the variable goodbye keeps the value it was assigned on line 2.

```
How many years old are you? 28
See you later! Sorry that you are not eligible :(
```

But if the if-statement executes, the assignment statement on line 10 overwrites the value of the variable goodbye with a new message, which prints instead.

```
How many years old are you? 82
You are eligible for Medicare!
See you later! Congrats on the good news!
```

5.5 Nesting

What if we wanted to ask a follow-up question? If the user did end up being eligible for Medicare, we'd next like to ensure that she registered for Medicare within the appropriate time window. Failure to do so results in lifetime penalties that permanetly raise the price she will have to pay for certain Medicare services. Alternatively, if the user wasn't eligible for Medicare, then we already know the user doesn't need to register, so we shouldn't bother the user with any follow-ups. In short, we want to ask the user another question, but only if he or she was determined to be eligible after asking the first one.

To represent this logic in Python, we will certainly need another if-statement for our follow-up question. But Python should encounter this second if-statement only if the first one executed. To ensure this happens, we need to put the second if-statement *inside* the first one. The block within this inner (or nested)

if-statement needs to be further indented by another four spaces in order to distinguish it from the outer if-statement.

```
1  # Ask for the user's age.
2  age = int(input('How many years old are you? '))
3
4  # If the user is older than 65, he or she is eligible .
5  if age >= 65:
6
7      # Did you register within the appropriate window?
8      on_time = input('Did you register in the 6 months around turning 65?')
9      if on_time != 'y':
10
11         # The user might have to pay a penalty.
12         print('You may have to pay a lifetime penalty :( ')
13
14     print('Those are all the questions I have for you.')
15
16 print('Have a nice day!')
```

If the user provides an age < 65 to the question on line 2 (meaning she is ineligible for Medicare), the guard of the outer if-statement on line 5 will be False , causing Python to skip the if-statement's entire body (the indented statements from lines 6 to 14) and go straight to line 16 (the next unindented statement).

Alternatively, if the user provides an age $>= 65$ to the question on line 2 (confirming that she is eligible for Medicare), the body of the outer if-statement (the code on lines 7 through 14) will execute.

First, the program will ask the follow-up question about whether the user registered in the appropriate time frame as on line 8. Using this answer, Python will evaluate the boolean test in the inner if-statement on line 9. If the user answered anything other than 'y' to the follow-up question, then the test will evaluate to True and the body of the inner if-statement (lines 11 and 12) will execute. If not, Python will skip lines 11 and 12. Either way, the print statement on line 14 will conclude the outer if-statement.

Once back on the outermost level of the program, Python will print the farewell message on line 16 and the program will end.

Here are a few possible executions of the program. If the user answers 24 to the first question, neither if-statement will execute, and the program will wrap up quickly.

```
How many years old are you? 24
Have a nice day!
```

If the user instead answers 74 to the first question, the program will execute the outer if-statement and ask a second question. If the user answers 'y' to that question, the inner if-statement will be skipped.

```
How many years old are you? 74
```

```
Did you register in the 6 months around turning 65? y
Those are all the questions I have for you.
Have a nice day!
```

If, instead, the user answers 'n' to the second question, indicating that she did not register in the six months around turning 65, the inner if-statement will execute as well.

```
How many years old are you? 74
Did you register in the 6 months around turning 65? n
You may have to pay a lifetime penalty :(
Those are all the questions I have for you.
Have a nice day!
```

In summary, the inner if-statement can execute only if the outer if-statement does. To answer a lingering question from earlier, this is why Python is so picky about indentation—it needs to be able to tell the difference between the code blocks for the outer and inner if-statements. Python requires that the amount of indentation be identical for all lines of code in the same block, which makes distinguishing blocks from one another much easier.

Note that Python is not limited to just two levels of nesting. The nested if-statement could itself contain another if-statement.

5.6 Else-Statements

Often, a program should have one behavior if the condition of an if-statement is True and a different behavior if it is False. For example, if the user is 65 or older, then the program should respond that she is eligible for Medicare. Otherwise, the program should respond with a different message, perhaps initiating a set of questions to determine whether the user meets other eligibility criteria for Medicare.

```
How many years old are you? 74
You are eligible for Medicare!
Have a nice day!
```

```
How many years old are you? 47
You may be eligible if you have a disability .
Have a nice day!
```

One way of achieving this behavior is to use a second if-statement that tests for the opposite condition.

```
1 # Ask for the user's age.
2 age = int(input('How many years old are you? '))
3
4 # If the user is older than 65, he is eligible .
5 if age >= 65:
```

```
6       print('You are eligible for Medicare!')
7
8   # Code to execute if the user is under 65
9   if age < 65:
10      print('You may be eligible if you have a disability.')
11
12  # A polite farewell.
13  print('Have a nice day!')
```

Since the conditions of the two if-statements are mutually exclusive, exactly one of them will always execute. Although this logic is sound, it is cumbersome and error-prone in practice. To make this code work properly, you will have to carefully scrutinize the logical properties of any boolean tests you write. Even if you do so perfectly, anyone who reads your code will be puzzled without the help of a long comment.

To make life easier, Python provides a simpler way of saying the same thing: an else-statement. Following any if-statement, you can simply write the keyword else and a colon to demarcate a code block you want to execute only when the if-statement doesn't. For example, we could replace the second if-statement above with an else-statement without changing the behavior of the program.

If an if-statement's boolean condition is True, the code in the if-statement executes and the corresponding else-statement is skipped. If the condition is False, the code in the if-statement is skipped and the else-statement executes. If-statements and else-statements come in pairs—the else-statement must immediately follow the if-statement to which it corresponds. The else keyword is indented at the same level as its partner if keyword. Just like an if-statement, an else-statement gets its own indented code block.

Rewriting the earlier code (the only change is on line 9):

```
1   # Ask for the user's age.
2   age = int(input('How many years old are you? '))
3
4   # If the user is older than 65, he is eligible.
5   if age >= 65:
6       print('You are eligible for Medicare!')
7
8   # Code to execute if the user is under 65.
9   else:
10      print('You may be eligible if you have a disability.')
11
12  # A polite farewell.
13  print('Have a nice day!')
```

If the user answers with an age of at least 65 to the question on line 2, the if-statement on lines 5 and 6 will execute. When it is done, Python skips the else-statement and proceeds to line 12. Alternatively, if the user answers with an age less than 65, Python skips the if-statement (since the boolean guard in

the if-statement is False) and proceeds to the else-statement on lines 9 and 10. After line 10, the program continues to line 12.

Just as before, if-statements can be nested inside else-statements. Zooming in on the else-statement on line 9 from above, we could add a follow-up question to handle the case where the user is eligible for Medicare at a younger age due to a disability.

```
8  # Code to execute if it the user is under 65.
9  else:
10     disabled = input('Are you disabled under soc. security? ')
11     if disabled == 'y':
12         print('You are still eligible for Medicare!')
13     else:
14         print('Sorry, you are ineligible .')
15
16 # A polite farewell.
17 print('Have a nice day!')
```

If the user is determined to be too young by the question on line 2 and the if-statement that follows on line 3, the program will go to the else-statement on line 9. In the updated version of that else-statement above, the program would then ask another question—whether the user is considered to be disabled according to social security. This condition is tested in the nested if-statement on line 11. That if-statement, in turn, has its own else-statement (line 13) for the event where the user answers 'n' to this second question.

So if the user is younger than 65 and does not have a qualifying disability, the program will execute as follows:

```
How many years old are you? 42
Are you disabled under soc. security? n
Sorry, you are ineligible .
Have a nice day!
```

Since the user answered 42 to the first question, the user is too young for Medicare and the program enters the outer else-statement on line 9. Based on the 'n' answer to the second question, the program subsequently enters the else part of the nested if-statement (line 13).

Alternatively, if the user answered 'y' to the second question, then the program would enter the nested if-statement on lines 11-12 and skip its else-block.

```
How many years old are you? 42
Are you disabled under soc. security? y
You are still eligible for Medicare!
Have a nice day!
```

If we wanted to, we could add an arbitrary number of nested if-statements over and over again for additional follow-up questions. We could enhance our program by following questions about eligibility with questions and advice about

registration. Our program will have two parts. First, assess whether a user is eligible, saving the result to a boolean variable rather than printing our decision directly as before.

```
# Ask for the user's age.
age = int(input('How many years old are you? '))

# If the user is older than 65, he is eligible .
if age >= 65:
    eligible = True

# Code to execute if the user is under 65.
else:
    disabled = input('Are you disabled under soc. security? ')
    if disabled == 'y':
        eligible = True
    else:
        eligible = False
```

This code functions exactly like the program we designed before, except that the print statements have been replaced with a boolean variable that keeps track of eligibility. We can now use this information to inform recommendations about registration.

```
# Ask about registration only if the user is eligible .
if eligible :
    print('You are eligible for Medicare!')

    # Check if the user registered soon enough.
    registered = input('Did you register in the 6 months around turning 65?
        ')
    if registered == 'y':
        print('You are good to go!')

    # Otherwise, did the user need to register that soon?
    else:

        # Is the user still employed?
        employed = input('Are you still employed full-time? ')
        if employed == 'y':

            # Is the user insured?
            insured = input('Are you insured through work? ')
            if insured == 'y':
                print('You do not need to register yet,')
                print('but you get Part A benefits for free.')

```

```
38              # If not, the user pays a penalty.
39              else:
40                  print('You will pay a penalty.')
41
42          else:
43              print('You will pay a penalty.')
44
45  print('Have a nice day!')
```

Lines 16 through 45 work off of the first 15 lines of code. If the user was determined to be eligible before, then the variable `eligible` will be set to True. Since the value of this variable is the guard for the if-statement on line 17, we will execute lines 20 to 43 only if the user was, indeed eligible; otherwise, Python skips from line 17 all the way to line 45 (which happens to be the last line of the program).

If the user is eligible, Python enters the if-statement on line 17 and a message confirming this eligibility is printed (line 18). Next, the program asks whether the user registered for Medicare in the six months prior to turning 65 (line 21); this window is the only time that a retiree can register for Medicare without incurring a lifetime penalty that makes collecting benefits more expensive. If the user answered `'y'`, then the guard of the nested if-statement on line 22 is True, causing the program to confirm that the user is in good shape. Since the if-statement executed, the corresponding else-statement (lines 26 to 43) is skipped, causing Python to immediately jump to line 45, the last line of the program.

Even if the user did not register on time, she can still avoid paying a penalty if she continues to work full-time for an employer who provides health insurance; if she does, she can register penalty-free when she retires (or is otherwise no longer employed and receiving benefits). If the user did not register on time (meaning she answered something other than `'y'`) to the prompt on line 21, the else-statement on line 26 will execute. This else-statement asks the user another question—whether the user is still employed full-time—on line 29. Line 30 contains a further nested if-statement: if the user is indeed employed, the guard of the if-statement on line 30 is True, causing Python move to the code block on lines 33 to 40. The program asks one final question on line 33: does the user get health insurance through work? If so, one last nested if-statement on line 34 informs the user that she does not yet need to register.

Each of these nested if-statements asking about employment and health insurance benefits through work has its own corresponding else statements on lines 42 and 39 respectively. Should either of these nested if-statement tests fail, the user will be informed that she will have to pay a penalty.

One sample execution for a user who is above retirement age but continues to work is below:

```
How many years old are you? 68
You are eligible for Medicare!
Did you register in the 6 months before turning 65? n
```

```
Are you still employed full-time? y
Are you insured through work? y
You do not need to register yet,
but you get Part A benefits for free.
Have a nice day!
```

5.7 Elif-Statements

Notice that the previous program involved a substantial amount of nesting. It tended to have a recurring pattern: if the answer to a question was "no", then the program had an else-statement that asked a follow-up question in a nested if-statement. If the answer to that follow-up question was also "no", another else-statement was followed by another nested if-statement as the program asks you about every possible case. Eventually, this nesting gets so deep that your code becomes confusing to look at—it becomes a case of bad style. Yet these sorts of repeated questions are common in both law and programming. "Are you still insured through work?" "No." "Do you have nearby providers who take Medicare?" "No." "Do you have a good Medicare Advantage plan in your locality under Medicare part C?" "Yes!"[1]

```
1  work = input('Do you have insurance through work? ')
2  providers = input('Do you have providers who take Medicare? ')
3  advantage = input('Do you have a good Medicare Advantage Plan nearby? ')
4
5  if work == 'y':
6      print('You do not need Medicare benefits yet.')
7  else:
8      if providers == 'y':
9          print('Do not enroll in Medicare Part C.')
10     else:
11         if advantage == 'y':
12             print('Consider enrolling in Medicare Part C.')
13         else:
14             print('You may want to consider moving.')
15
16 print('Goodbye.')
```

[1] For context, Medicare Advantage (Medicare Part C) is a health insurance network run by private insurance companies as either a Health Maintenance Organization (HMO), a Preferred Provider Organization (PPO), or a Private Fee For Services (PFFS) plan, any of which is an integrated health care plan. A Medicare Advantage plan must have at least the benefits provided by Medicare Parts A and B. Enrolling in Medicare Part C restricts your ability to use health care providers to those within the plan, may cost less than Medicare Part B, has large out-of-network costs, and is determined by your locality. It is generally a good option for participants who live in areas where few health care providers accept Medicare. The drawback of Medicare Part C is that, after a one-year trial period in a Medicare Advantage Plan, the participant must pay a penalty to revert to Medicare Parts A and B.

Thankfully, Python provides a way around messily trying to nest if-statements within else-statements as we did above: elif-statements. An elif-statement is an if-statement and an else-statement combined into one (elif is a portmanteau of "else" and "if"). It comes after an if-statement and contains its own code block, just like an else-statement. But it has its own boolean test, just like an if-statement.

Here's how an elif-statement works. When you create an if-statement, you can follow it with one or more elif-statements and then optionally end with a final else-statement. First, Python will try out the if-statement's boolean test. If it is True, then the code in the body of the if-statement will execute and all of the elifs and the else will get skipped. If the if-statement's boolean test is False, however, Python will proceed to the first elif-statement and try that test. The elif works just like an if-statement. If its test is True, its body executes and all other elifs and elses are skipped; if its test is False, Python tries the next elif and so on. If Python has tried every elif-statement unsuccessfully and you have written a concluding else-statement, Python will execute that; otherwise, it will simply return to the normal flow of execution.

As an example, we can rewrite the code above with elif statements.

```
1  work = input('Do you have insurance through work? ')
2  providers = input('Do you have providers who take Medicare? ')
3  advantage = input('Do you have a good Medicare Advantage Plan nearby? ')
4
5  if work == 'y':
6      print('You do not need Medicare benefits yet.')
7  elif providers == 'y':
8      print('Do not enroll in Medicare Part C.')
9  elif advantage == 'y':
10     print('Consider enrolling in Medicare Part C.')
11 else:
12     print('You may want to consider moving.')
13
14 print('Goodbye.')
```

This code is functionally identical but far easier to read. First, on line 5, Python will check whether the variable work is equal to the string `'y'`. If it is, the code on line 6 will execute, after which Python will skip all of the other elifs and elses and go to line 14. Alternatively, if this test fails, Python will try the elif on line 7 and check whether providers is equal to `'y'`. If it is, the code on line 8 will execute, after which Python will skip to line 14. If it isn't, Python will try the elif on line 10. If that test fails, it will execute the body of the else on line 12. But no matter what, it will continue the main flow of execution on line 14 when all is said and done.

Just as with if-statements, you can include any Python code you like in elif-statements and else-statements. For example, we could rewrite the program above to store the text that would be printed in a string and print it at the very end instead.

```
1  work = input('Do you have insurance through work? ')
2  providers = input('Do you have providers who take Medicare? ')
3  advantage = input('Do you have a good Medicare Advantage Plan nearby? ')
4
5  response = 'You may want to consider moving.'
6
7  if work == 'y':
8      response = 'You do not need Medicare benefits yet.'
9  elif providers == 'y':
10     response = 'Do not enroll in Medicare Part C.'
11 elif advantage == 'y':
12     response = 'Consider enrolling in Medicare Part C.'
13
14 print(response)
15 print('Goodbye.')
```

On line 6, the program creates a variable called response, which stores the proper response-text for the recommendation that the program will make to the user. Initially, it is assigned a default value (line 6). The if-statement and elif-statements overwrite the variable with the appropriate reaction for each scenario. This variable is printed after the if-statement is done (line 14). Notice that, by restructuring the program in this way, we have eliminated the need for an else-statement. If none of the if and elif tests are True, response still has the default value it was assigned at the very beginning—the value we needed to set before in an else-statement.

When working with elif-statements, one common mistake is to write if instead of elif .

```
1  work = input('Do you have insurance through work? ')
2  providers = input('Do you have providers who take Medicare? ')
3  hmos = input('Do you have HMOs who take Medicare? ')
4
5  if work == 'y':
6      print('You do not need Medicare benefits yet.')
7  if providers == 'y':
8      print('Do not enroll in Medicare Part C.')
9  if hmos == 'y':
10     print('Consider enrolling in Medicare Part C.')
11 else:
12     print('You may want to consider moving.')
13
14 print('Goodbye.')
```

If you did this as in the program above, you would get some strange results.

Do you have insurance through work? *y*
Do you have providers who take medicare? *y*
Do you have a good Medicare Advantage Plan nearby? *n*

```
You do not need Medicare benefits yet.
Do not enroll in Medicare Part C.
You may want to consider moving.
Goodbye.
```

How did the program manage to print the text on lines 6, 8, *and* 12? As far as Python is concerned, each of these if-statements is completely separate, so Python will handle them individually. When the test on line 5 succeeds, Python will enter the if-statement on lines 5-6 and print the text on line 6. Afterward, it will proceed to the next line after this if-statement—line 7. Since the condition on line 7 is also True, Python will print the text on line 8 and resume normal execution, moving to the final if-statement on line 9. When this test fails, Python will look for a corresponding else-statement, which it finds on line 11. It therefore prints the message on line 12 before exiting the else-statement and going to line 14. It may seem subtle, but there is a world of difference between separate if-statements and one large if-statement with several elifs attached. Each of these if-statements is distinct, with the else-statement attached to the if-statement that immediately precedes it.

If you insisted on truly avoiding all elif-statements, you would need to use a complicated boolean test to protect the text that is printed in the else-statement. Doing so is possible but clearly far from ideal:

```
1  work = input('Do you have insurance through work? ')
2  providers = input('Do you have providers who take Medicare? ')
3  advantage = input('Do you have a good Medicare Advantage Plan nearby? ')
4
5  if work == 'y':
6      print('You do not need Medicare benefits yet.')
7  if (work != 'y') and (providers == 'y'):
8      print('Do not enroll in Medicare Part C.')
9  if (work != 'y') and (providers != 'y') and (advantage == 'y'):
10     print('Consider enrolling in Medicare Part C.')
11 else:
12     print('You may want to consider moving.')
13
14 print('Goodbye.')
```

The boolean condition ensures that the text on line 9 is printed only when the guards of all of the other if-statements were False

As one final example of the massive difference between using elif-statements and if-statements, consider the program below, which determines the fraction of social security benefits that you receive if you retire early:

```
1  age = int(input('At what age do you plan to retire? '))
2
3  if age < 62:
4      print('You are too young to collect benefits.')
5  elif age < 63:
```

```
6       print('75% benefits.')
7   elif age < 64:
8       print('80% benefits.')
9   elif age < 65:
10      print('86.7% benefits.')
11  elif age < 66:
12      print('93.3% benefits.')
13  else:
14      print('66 is full retirement age - 100% benefits.')
```

This program relies on the fact that, as soon as one if or elif (known as branches of the if-statement) triggers, the rest are skipped. So if you are 63 years old, the first if and elif are False, the second elif executes, and the rest are skipped (even though all of their tests would be True if they were ever evaluated). In that case, the user would see the following result:

```
At what age do you plan to retire? 63
80% benefits.
```

Had you mistakenly used five if-statements rather than one if-statement accompanied by four elif-statements, you would have seen a very different outcome:

```
At what age do you plan to retire? 63
80% benefits.
86.7% benefits.
93.3% benefits.
```

Why did every almost every possible line of text print? Let's take a look at the program.

```
1   age = int(input('At what age do you plan to retire? '))
2
3   if age < 62:
4       print('You are too young to collect benefits.')
5   if age < 63:
6       print('75% benefits.')
7   if age < 64:
8       print('80% benefits.')
9   if age < 65:
10      print('86.7% benefits.')
11  if age < 66:
12      print('93.3% benefits.')
13  else:
14      print('66 is full retirement age - 100% benefits.')
```

The first two if-statement tests will fail, but all of the rest will pass. So when Python gets to line 7, the test passes and it will print the text on line 8. Afterward, it will return to the normal flow of execution...on line 9, where it will find another if-statement whose test will also pass. The final if-statement will

also pass, printing all of the text except that about full retirement age, which is skipped over because it is in an else-statement whose corresponding if-statement (line 11) executed successfully.

5.8 Conclusion

In this chapter, you have learned your first form of control flow, the larger set of Python statements that let you shape the way programs execute. You also saw why boolean expressions, which initially seemed pointless, have a central role to play in the smooth operation of control-flow.

If-statements give you the ability to write programs that branch, choosing to conditionally execute or skip certain blocks of code. This is a powerful addition to your Python toolkit that dramatically increases the range of programs you are able to write. In the next chapter, you will learn to write programs that repeat code, making it possible to handle uncertain amounts of input and execute for an indefinite period of time (perhaps even forever).

5.9 Cheatsheet

If-Statements

(Note that the phrases within angle brackets (< and >) should be substituted for actual Python code.)

```
1  if <boolean expression>:
2      <line of code within if-statement>
3      <optionally, another line of code within if-statement>
4
5  <line of code after if-statement>
6  <another line of code after if-statement>
```

An if-statement has four components.

1. It begins with the keyword `if`.
2. A boolean expression (a guard).
3. A colon.
4. One or more lines of code indented to the same level (a code block).

When Python encounters an if-statement (line 1), it evaluates the boolean expression. If the boolean expression evalutes to True, Python executes the indented code block (lines 2-3) and then continues evaluating the program normally (lines 5, 6, and beyond). If the boolean expression evaluates to False, Python skips the indented code block and picks up at the next line indented to the same level as the `if` operator. (line 5).

Else-Statements

```
1  if <boolean expression>:
2      <line of code within if-statement>
3      <optionally, another line of code within if-statement>
4  else:
5      <line of code within else-statement>
6      <optionally, another line of code within else-statement>
7
8  <line of code after if-statement>
9  <another line of code after if-statement>
```

An else-statement is optional. If it is included, it must directly follow an if-statement and its associated code block.

If the if-statement's boolean expression (line 1) evaluates to True, Python executes its indented code block (lines 2-3), skips the else-statement (line 4) and its indented code block (lines 5-6), and picks up at the next line indented to the same level as the `if` and `else` operators (line 8).

If the if-statement's boolean expression (line 1) evaluates to False, Python skips its indented code block (lines 2-3) and executes the else-statement's indented code block (lines 5-6) instead. Afterwards, it picks up at the next line indented to the same level as the if and else operators (line 8).

Elif-Statements

```
1  if <boolean expression>:
2      <line of code within if-statement>
3      <optionally, another line of code within if-statement>
4  elif <boolean expression>:
5      <line of code within elif-statement>
6      <optionally, another line of code within elif-statement>
7  elif <boolean expression>:
8      <line of code within elif-statement>
9      <optionally, another line of code within elif-statement>
10 else:
11     <line of code within else-statement>
12     <optionally, another line of code within else-statement>
13
14 <line of code after if-statement>
15 <another line of code after if-statement>
```

In its full form, an if-statement contains one if-block (lines 1-3) optionally followed immediately by one or more elif-blocks (lines 4-6 and lines 7-9) and optionally ended with one else-block (lines 10-12).

If the if-statement's boolean expression (line 1) evaluates to True, Python executes its indented code block (lines 2-3), skips the elif-blocks (line 4-6 and lines 7-9) and the else-block (lines 10-12) and picks up at the next line indented to the same level as the if operator (line 14).

If the if-statement's boolean expression (line 1) evaluates to False, Python skips its indented code block (lines 2-3). It then tries the first elif-block's boolean expression (line 4). If it evaluates to True, Python executes the corresponding code block (lines 5-6), skips the remaining elif-blocks and the else-block, and picks up at the next line indented to the same level as the if operator (line 14).

If the boolean conditions in the if-statement and first elif-block both evaluate to False, Python will try each elif-block in succession until one of their boolean conditions evaluates to True. Should none of them evaluate to True, it will evaluate the else-statement's code block (lines 11-12) and then pick up at the next line indented to the same level as the if operator (line 14). If none of the elifs evaluate to False and there isn't an else-statement, Python will proceed directly to line 14 without evaluating any of the blocks.

Nested If-Statements

```
1  if <boolean expression>:
2      <line of code within if-statement>
3
```

```
4       if <boolean expression>:
5           <line of code within nested if-statement>
6           <another line of code within nested if-statement>
7
8       <line of code after nested if-statement>
9
10  <line of code after if-statement>
11  <another line of code after if-statement>
```

If-statements can be contained (nested) within other if-statements (or elif-blocks or else-blocks). The inner if keyword (line 4) is indented at the same level as the other statements inside the outer if-statement (e.g., lines 2 and 8).

If the outer if-statement (line 1)'s boolean expression is False, Python skips its entire code-block (lines 2-8) and proceeds directly to the next line indented at the same level as the if operator (line 10).

If the outer if-statement (line 1)'s boolean expression is True, Python executes its code block (lines 2-8). When it encounters the nested if-statement (line 4), it tests its boolean expression. If the nested boolean expression evaluates to True, Python evalutes its code block (the doubly indented lines 5 and 6) and picks up at the next line indented to the same level as the nested if operator (line 8), executing the remainder of the outer if-statement's body. If the nested boolean expression evaluates to False, Python skips its code block (the doubly indented lines 5 and 6) and proceeds directly to line 8, executing the remainder of the outer if-statement's body.

If statements can be nested arbitrarily deep. Nested if-statements can have their own elif-blocks and else-blocks that should be indented to the same level as the if operator to which they correspond.

Keywords

Body—The body if an if, elif, or else statement is the code blockthat it guards. Example: the body of an if-statement is the code block that Python executes if the if-statement's guard evaluates to True.

Branch—Code is said to branch when the program's execution follows one of many possible paths it might have taken. Example: a program branches when it determines whether to execute an if-statement or its corresponding else-statement. Each of these separate possible paths is known as a branch.

Code Block—A series of one or more lines of code that are indented at the same level (or deeper levels in the case of nesting). Python either executes the entirety of a code block or executes none of a code block. Example: the indented code guarded by an if-statement is a code block. If the if-statement's guard is True, the entire code block executes; if it is False, the entire code block is skipped.

Condition—See: Guard.

Conditional—Code that is executed only if certain conditions are met, for example that a variable takes on a particular value or range of values.

Control Flow—The process of conditionally modifying the order in which Python executes a program.

Guard—The boolean expression that determines whether Python will execute the body of an if-statement or elif-statement.

Nesting—When one if-statement is contained inside the body of another if-statement.

Test—See: Guard.

Chapter 6

While Loops

Until you learned about if-statements, executing a Python program was straightforward: just evaluate each line of code in order from top to bottom. In the previous chapter, we modified this strategy slightly, giving you the power to skip large swaths of code based on boolean conditions. Even then, though, every program you wrote had a finite lifetime—Python always ran out of code eventually, bringing your program to an end.

In this chapter, we will break that paradigm, making it possible to repeat blocks of code so long as a boolean condition holds, giving you the ability to write programs that run for an indefinite amount of time—or even forever. Your programs will be able to take arbitrary amounts of input, like a list of names or a ledger of bank transactions, and process them in ways that require repetition. You will do so through the use of loops, which are similar in appearance and behavior to if-statements. Although Python has only one kind of if-statement, it has two different types of loops. In this chapter, we will focus on the more generic kind—while-loops—leaving for-loops for later.

6.1 Basics

A while-loop is very closely related to an if-statement. Both contain a block of code that is executed if a boolean condition is True. The only difference is that, after the body of an if-statement is evaluated, Python picks up where it left off, executing the code that comes afterward. In contrast, a while-loop repeats its block of code over and over again so long as its boolean condition is True. Only after the condition becomes False does Python resume moving downward through your program normally.

Consider the following program, which contains an if-statement.

```
1  count = 0
2
3  if count < 5:
4      print('Inside the if-statement, count is ' + str(count))
```

```
5       count = count + 1
6
7   print('After the if-statement, count is ' + str(count))
```

After last chapter, this code should be quite familiar. On line 1, a variable count is created and initialized to 0. On line 3, an if-statement guards the code on lines 4 and 5 with a boolean condition. Specifically, the code inside the if-statement will execute if the value of count is less than 5 (which it clearly is). The code block inside the if-statement prints the value of count and then adds one to its value. Finally, the statement on line 7, which is outside of the if-statement, will print the value of count at the end of the program.

When this program is executed, the result shouldn't surprise you.

```
Inside the if-statement, count is 0
After the if-statement, count is 1
```

The variable count starts with the value 0, which is printed and incremented to 1 in the if-statement. Finally, it is printed again with its new value after the if-statement is done.

We can convert this code into a while-loop with a very simple modification: replace the keyword if with the keyword while. Everything else can stay the same. A while-loop looks just like an if-statement aside from this substitution—it contains the word while followed by a boolean condition, a colon, and an indented code block. The modified program appears below. The only changes are swapping in the while keyword and modifying the printed text to match this alteration.

```
1   count = 0
2
3   while count < 5:
4       print('Inside the while-loop, count is ' + str(count))
5       count = count + 1
6
7   print('After the while-loop, count is ' + str(count))
```

Here's how a while-loop works. Just like an if-statement, Python will arrive at the beginning of the while-loop (line 3 in the example above) and test whether the boolean condition is True. If it isn't, Python will skip the body of the while-loop and continue with the normal flow of the program (jumping to line 7 above) just like an if-statement.

If the while-statement's boolean guard is True, however, Python will execute the indented code block that follows (lines 4 and 5 above). So far, this is still just like an if-statement. The difference is that, once the code block is done executing, Python will go back to beginning of the loop (line 3) and test the boolean condition again. If it is still True, Python will execute the indented code again, return to the beginning, test again, and keep repeating as long as the condition continues to be True. Each execution of the body of the loop is called an iteration. If ever the condition becomes False, Python will stop

looping, skip the body of the loop, and continue with the normal flow of the program (line 7) just like an if-statement.

When executed, the above program will print the following output.

```
Inside the while-loop, count is 0
Inside the while-loop, count is 1
Inside the while-loop, count is 2
Inside the while-loop, count is 3
Inside the while-loop, count is 4
After the while-loop, count is 5
```

Initially, the value of count is 0 (as established on line 1). On line 3, the while keyword instructs Python to test the boolean condition that follows—to check whether count is less than 5. Since it is, the code inside the loop executes, printing the value of count (at this point in time, 0) and then increasing the value by 1.

Once the body of the loop is done executing, Python returns to line 3 and checks the boolean condition again. Since the value of count is 1 (which is still less than 5), the body of the loop executes again, printing count and increasing its value to 2. This process continues three more times, printing that count is 2, 3, and 4.

On the fifth time through the loop, the value of count is increased from 4 to 5 on line 5. When Python subsequently returns to the while-loop test on line 3, it finds that count is no longer less than 5 (since it now equals 5). At this point, Python skips the body of the loop and continues with the program, executing the print statement on line 7 and bringing the program to an end.

Another way of understanding a while-loop is to think of it as an if-statement that is copied and pasted over and over again in your program.

```
1  count = 0
2
3  if count < 5:
4      print('Inside the while-loop, count is ' + str(count))
5      count = count + 1
6
7  if count < 5:
8      print('Inside the while-loop, count is ' + str(count))
9      count = count + 1
10
11 if count < 5:
12     print('Inside the while-loop, count is ' + str(count))
13     count = count + 1
14
15 # Keep copying and pasting the if-statement until the boolean
16 # test comes back False.
17 etc.
18 etc.
```

```
19
20 print('After the while-loop, count is ' + str(count))
```

The code written above isn't valid Python (etc. is unfortunately not a valid Python command), but it is a good mental model for the way while-loops work. A while-loop is really just an if-statement that keeps executing as long as its boolean test continues to be True.

6.2 Mortgage Calculator

6.2.1 Designing a Mortgage Calculator

Now that you have a basic sense for how while-loops work, let's consider another example: a mortgage calculator. Suppose we want to write a program that calculates the length of time it will take to pay off a mortgage and the total amount of interest that will be paid.[1] The program will need three pieces of information that it will request using the input function: the initial balance, the APR, and the monthly payment. In the example below, the program calculates these figures for a $500 mortgage at an 8% interest rate where the payment is $10 a month.

```
$ python3 loan_calculator.py
Loan amount: $500
Interest rate: 8
Monthly payment: $10
Months to pay loan: 62
Total interest paid: $110.22346688955415
```

Before we can start writing the program, here's a quick primer on how mortgages are calculated. You begin with an initial loan amount, in this case $500. Every month, you make a payment, in this case $10. Part of that payment goes toward paying that month's interest; whatever is left after paying interest gets subtracted from the loan amount. To compute the monthly interest, calculate the annual interest of the loan amount and divide by 12. (Since there are 12 months in a year, dividing by 12 converts the annual interest into the monthly interest).

Let's continue our example. In the first month, the loan amount is $500. Since the annual interest rate is 8%, the annual interest would be 8% of $500 (the current balance), or $40. We convert this number into a monthly value by dividing by 12, which equals $3.33. This is the interest that gets paid during the first month of the loan. Since each monthly payment is $10, this leaves $6.67

[1] This program is really more applicable to student loans. A typical mortgage has a fixed term. If we were considering mortgages, it would be more natural to calculate the monthly payment from the loan amount, interest rate, and term. Student loans have no such fixed term, so it makes sense to calculate the time it would take to pay off a student loan given the loan amount, interest rate, and monthly payment. We avoid the student loan terminology here because it is likely to be a sore subject for many of our readers.

of the payment that is used for paying down the overall loan amount, leaving $493.33 as the loan amount for next month.

At the beginning of the second month, we run the same calculation. After the first month's payment, the loan amount is now $493.33. The annual interest is 8% of this amount, or $39.47. Dividing by 12, that means this month's interest payment is $3.28. Notice that, since we paid off a small amount of the loan last month, the amount of interest we have to pay this month on the now-lower loan amount is also a little smaller. Subtracting the monthly interest of $3.28, the remaining $6.72 of the $10 monthly payment goes toward paying down the loan amount. This leaves $486.61 as the loan amount for the next month.

This process repeats every month until the loan is paid off.

To encode this logic in a program, we first need to request the initial loan amount, the interest rate, and the monthly payment. We do so using the input function.

```
1  # Request the loan amount, interest rate, and monthly payment.
2  balance = float(input('Loan amount: $')
3  rate    = float(input('Interest rate:  ')) / 100
4  payment = float(input('Monthly payment: $'))
```

Notice that, on line 3, rate divides the command line argument by 100 to convert a percentage (i.e., 8%) into a decimal (i.e., 0.08).

Next, we will create the main while-loop. Each iteration of the loop will simulate one month of time going by. It will need to calculate the interest to be paid, subtract it from the monthly payment, and deduct what remains from the overall balance. When should the loop stop? In concrete terms, what should its boolean condition be? It should stop when the balance reaches 0, indicating that the loan has been paid off. The proper boolean condition, then, is

balance > 0

As soon as this condition is violated, the loan is fully paid. Since the we want the loop to keep track of the number of months it took to pay the mortgage and the total interest that was paid, we will need to declare two variables beforehand.

```
6  # Initialize summary variables.
7  months = 0
8  total_interest = 0
```

During each iteration of the loop, we will need to increase months by 1 to represent the passage of time and total_interest by the amount of interest paid during that month. We are now ready to write the loop itself.

```
10 while balance > 0:
11     # Move time forward.
12     months = months + 1
13
14     # Calculate the interest portion of the payment.
15     interest = rate * balance / 12
```

```
16      total_interest += interest
17
18      # Deduct the remaining monthly payment from the balance.
19      payment_less_interest = payment - interest
20      balance = balance - payment_less_interest
```

The while statement on line 10 includes the boolean test we settled on previously. It will continue looping so long as the answer to the question, "is balance bigger than 0?" is "yes." Time ticks forward by one month on line 12. Line 15 calculates the total interest, which is the annual interest (the interest rate multiplied by the current loan balance) divided by 12 (since we're only calculating interest for one month). This amount is added to the running total of interest paid on line 16. On line 19, the program calculates the portion of the monthly payment that will go toward reducing the loan balance. This amount is subtracted from the loan balance on line 20.

This logic is everything we need to fill out the body of the loop. Line 21 is the last line in the loop's code block (that is, it is the last indented line). After line 21, Python will jump back to line 10, where it will check whether balance is still larger than 0. If it is, it will execute the body of the loop again, advancing time forward another month and making another loan payment. When the balance finally does reach 0 or below, the test on line 10 will be False and Python will exit the loop, picking up on the next line of code indented to the same level as the while keyword. In this case, that means line 22 below.

```
22  # Display the output to the user.
23  print('Months to pay mortgage: ' + str(months))
24  print('Total interest paid: $' + str(total_interest))
```

After the loop, lines 21 and 22 print the program's output: the number of months required to pay off the loan and the total interest paid in the process of doing so.

6.2.2 Nesting

A loop's indented code block behaves in the same way as an if-statement's indented code block. You can include any statements you like inside, including nested if-statements and while-loops that have their own code blocks. In turn, these nested if-statements and while-loops can have their own nested code blocks, just as in the previous chapter. Later, we will explore putting loops inside other loops. Suppose that you have taken out a variable-rate mortgage whose rate will increase after the first five years. We can modify the loan calculator program to handle this kind of loan using an if-statement nested within the loop.

```
1  # Request the loan amount, interest rates, and monthly payment.
2  balance = float(input('Loan amount: $'))
3  initial_rate = float(input('Initial rate: ')) / 100
4  eventual_rate = float(input('Eventual rate: ')) / 100
```

```
5  payment = float(input('Monthly payment: $'))
6
7  # Initialize summary variables.
8  months = 0
9  total_interest = 0
10
11 while balance > 0:
12     # Move time forward.
13     months = months + 1
14
15     # Use initial rate if loan is < 5 years old.
16     if months < 60:
17         rate = initial_rate
18     else:
19         rate = eventual_rate
20
21     # Calculate the interest portion of the payment.
22     interest = rate * balance / 12
23     total_interest += interest
24
25     # Deduct the remaining monthly payment from the balance.
26     payment_less_interest = payment - interest
27     balance = balance - payment_less_interest
28
29 # Display the output to the user.
30 print('Months to pay mortgage: ' + str(months))
31 print('Total interest paid: $' + str(total_interest))
```

The program has changed in two ways. First, it now accepts an additional input value (line 4) that specifies the interest rate that the loan jumps to after five years. Second, lines 16 to 19 determine which rate to use. The if-statement creates a variable, rate, that stores the proper interest rate for later use. If the loan is less five years (60 months) old, it uses the initial interest rate (line 17). If the loan is more than five years old, it selects the second interest rate instead (line 19).

When Python encounters the nested if-statement on line 16, it executes either the if-block (lines 16-17) or the else-block (lines 18-19) just as in the previous chapter. When it finishes executing either line 17 or line 19 (depending on whether months is smaller than 60), Python resumes execution at line 22, the next line in the outer while-loop's code block (the next line indented to the same level as the if keyword).

6.2.3 Break

One feature we might consider adding is a way to detect when the loan will never be paid off—when the monthly interest exceeds the monthly payment.

To do so, we will need a way to exit a loop mid-stream. In other words, if the program detects that the loan can never be paid off, it shouldn't bother to attempt any payment calculations. Instead, it should print an error message and immediately exit the loop.

Python makes it possible to exit a loop early with the break command. If you type the statement break on its own line, Python will immediately exit the current loop, instantly jumping to the code that comes afterward. The break command offers a convenient way to exit a loop without having to wait for the loop to return to the beginning and test the boolean condition again.

```
10 while balance > 0:
11     # Move time forward.
12     months = months + 1
13
14     # Calculate the interest portion of the payment.
15     interest = rate * balance / 12
16     total_interest += interest
17
18     # Ensure that this monthly payment is sufficient to
19     # reduce the balance of the loan.
20     if interest >= payment:
21         print('You will never be able to pay off this loan.')
22         break
23
24     # Deduct the remaining monthly payment from the balance.
25     payment_less_interest = payment - interest
26     balance = balance - payment_less_interest
27
28 if balance <= 0:
29     print('Months to pay mortgage: ' + str(months))
30     print('Total interest paid: $' + str(total_interest))
```

Lines 18 to 22 are new. If the interest that needs to be paid in a given month is greater than the monthly payment, then the balance would increase every month and it would be impossible to ever pay off the loan. On line 21, the program prints a warning to this effect and then breaks out of the loop (line 22), jumping straight to the next line after the body of the loop (line 28). The final print statements (lines 29-30) are now enclosed in an if-statement to ensure that they only appear if the loan was fully paid off. Were they not guarded by the if-statement, then they would print even if the loop exited for the opposite reason—that the loan could never be paid off.

We can now experiment with monthly payments to find one that is too small to pay back a $1,000 loan at 5% interest.

```
$ python3 loan_calculator.py
Loan amount: $1000
Interest rate: 5
```

```
Monthly payment: $20
Months to pay mortgage: 57
Total interest paid: $123.69206860245193

$ python3 loan_calculator.py
Loan amount: $100
Interest rate: 5
Monthly payment: $8
Months to pay mortgage: 177
Total interest paid: $415.4988120347951

$ python3 loan_calculator.py
Loan amount: $1000
Interest rate: 5
Monthly payment: $4
You will never be able to pay off this loan.
```

Note that break only exits one level of looping. If you are in multiple, nested loops, break will only jump out of the inner-most loop. Python will resume the flow of execution in the next level outside of the inner-most loop, which may well be within an outer loop (or several). We will see examples of nested loops in later sections.

6.2.4 Continue

Closely related to the break command is continue, which skips the rest of the current iteration of the loop but does not exit. Instead, it returns to the boolean test at the beginning of the loop and immediately starts a new iteration. As an example of where continue might come in handy, consider a loan that promises no payments for the first six months. One way of representing this behavior would be to enclose most of the loop in an if-statement.

```
10 while balance > 0:
11     # Move time forward.
12     months = months + 1
13
14     # No payments for the first six months.
15     if months > 6:
16         # Calculate the interest portion of the payment.
17         interest = rate * balance / 12
18         total_interest += interest
19
20         # Deduct the remaining payment from the balance.
21         payment_less_interest = payment - interest
22         balance = balance - payment_less_interest
```

The parts of the program that manipulate interest and payments are enclosed in an if-statement that prevents them from running until six months have tran-

spired. This code is correct, but it is less than perfect from a style perspective—all but one statment of the loop is inside an if-statement. Code tends to be more difficult to read when there is a large amount of indenting, so it is generally better to reduce the amount of indenting where possible. Using continue, we can say express the same program more cleanly.

```
10 while balance > 0:
11     # Move time forward.
12     months = months + 1
13
14     # No payments for the first six months.
15     if months <= 6:
16         continue
17
18     # Calculate the interest portion of the payment.
19     interest = rate * balance / 12
20     total_interest += interest
21
22     # Deduct the remaining monthly payment from the balance.
23     payment_less_interest = payment - interest
24     balance = balance - payment_less_interest
```

When Python reaches line 15, it will test whether six or fewer months have transpired. If so, it will enter the body of the if-statement and arrive at line 16, which causes Python to immediately complete that iteration of the loop and jump to line 9 to begin the next one. In doing so, it skips all of the mortgage payment logic on lines 18-24, effectively skipping a month of payments. The only variable that gets updated is months on line 12, which is modified before the if-statement.

Like break, continue is rarely essential—there are typically other ways of reconfiguring the contents of a loop to avoid using either statement. But they are often convenient and result in far more streamlined code than many alternative options. You probably won't use these two commands often, but when you do you will be very grateful that the designers of Python were generous enough to include them.

6.3 Common Mistakes

Although you might think that counting is easy, loops will quickly convince you otherwise. There are a series of loop-related mistakes that are so common among even professional programmers that we think it is useful to list them here.

6.3.1 Useless Loops

Consider the following slight modification of the program we studied at the beginning of this chapter.

```
1  count = 0
2
3  while count > 5:
4      print('Inside the while-loop, count is ' + str(count))
5      count = count + 1
6
7  print('After the while-loop, count is ' + str(count))
```

The only change is that the $<$ operator on line 3 has been swapped for a $>$ operator. This program now instructs Python to loop as long as the value of count is *greater than* 5. Unfortunately, this will never be the case. Since the value of count is initially 0 (which isn't greater than 5), the loop's boolean test will fail the first time through, causing Python to skip straight to line 7 without ever executing the body of the loop. To confirm this summary, here is the output Python will produce:

```
After the while-loop, count is 0
```

Although the bug was small—just a single mistyped character—the change in the program's output was enormous. When you write a program involving a loop that doesn't seem to be running at all, the culprit is likely to be in the boolean condition.

6.3.2 Infinite Loops

Another slight modification of the program will lead to a dramatically different error.

```
1  count = 0
2
3  while count < 5:
4      print('Inside the while-loop, count is ' + str(count))
5      count = count - 1
6
7  print('After the while-loop, count is ' + str(count))
```

The boolean test has been fixed, but the + operator has been swapped for a − on line 5. On every iteration of the loop, this statement will subtract 1 from the value of count rather than increasing it.

Why is this a problem? The loop will never exit—count will always be less than 5!

```
Inside the while-loop, count is 0
Inside the while-loop, count is -1
Inside the while-loop, count is -2
```

```
Inside the while-loop, count is -3
Inside the while-loop, count is -4
Inside the while-loop, count is -5
Inside the while-loop, count is -6
...
```

This mistake is called an infinite loop, since the program will run forever if you leave it to do so. When you get stuck in an infinite loop, you can press the keyboard sequence ctrl+c (even on a Mac, it's still the ctrl key) to force the program to stop. Infinite loops are a rite of passage for beginners and a continual source of frustration for experts.

We could have caused the first version of the loan calculator program to enter an infinite loop by putting in a payment that is too small to cover the monthly interest. Since the monthly payment hadn't covered all of the accrued interest, the unpaid interest would get added to the loan balance, causing the loan balance to increase each month. Since the loan balance would never reach 0, the loop would iterate forever.

Another common source of infinite loops is neglecting to modify the value of count at all. Incrementing your loop's counter is easy to forget when you're working through the logic of a complicated loop.

```
1 count = 0
2
3 while count < 5:
4     print('Inside the while-loop, count is ' + str(count))
5
6 print('After the while-loop, count is ' + str(count))
```

Since the loop's boolean condition is initially True and the value of count doesn't change in the loop's body, the condition will never change either.

```
Inside the while-loop, count is 0
Inside the while-loop, count is 0
Inside the while-loop, count is 0
Inside the while-loop, count is 0
Inside the while-loop, count is 0
Inside the while-loop, count is 0
...
```

It is exceedingly common for a loop to have a counter variable that keeps track of the number of iterations in this fashion. In the loan calculator program, the variable months keeps track of the number of months of payments conducted so far, which happens to be the same as the number of loop iterations so far.

6.3.3 Off-By-One Errors

Even when your loop does manage to run at all and avoids running forever, making sure it runs the correct number of times is still a tricky endeavor. One

common mistake is to write a loop that runs one too many (or too few) times. Collectively, these sorts of bugs are known as off-by-one errors. No matter how many years you have been programming, you will continue to run into off-by-one errors.

Consider an example. Suppose you want to print the string `'hello'` four times. Just like before, you will need a variable that serves as a counter, keeping track of the number of times you have printed `'hello'` so that you stop after four. This counter should initially be set to 0, since the program hasn't printed `'hello'` at all.

```
1 # Number of hellos printed.
2 count = 0
```

The body of the loop itself is simple. It should print the string `'hello'` and then increase the value of count by 1.

```
3 while ???:
4     print('hello')
5     count = count + 1
```

The tricky part is deciding on the right boolean condition. Since we want it to print `'hello'` four times, we might try

```
3 while count <= 4:
```

However, when you run this code, you will find that it prints `'hello'` five times.

```
hello
hello
hello
hello
hello
```

The first three `'hello'`s print smoothly, increasing the value of count from 0 to 1, from 1 to 2, and from 2 to 3. After the fourth `'hello'` is printed, the value of count correspondingly increases from 3 to 4. When Python tries the loop's boolean test again at the beginning of the next iteration, it finds that count is still less than or equal to 4 (since count is exactly 4), so it prints `'hello'` a fifth time and increases the value of count to 5. Only at this point does the boolean test fail, exiting the loop and ending the program.

One way to fix this error would be to modify the boolean test to check whether count is less than or equal to 3 rather than 4.

```
3 while count <= 3:
```

This ensures that the last iteration of the loop takes place when count is 3, since count's value will change to 4 inside of the loop, at which point we have printed `'hello'` the proper number of times.

As a different way of saying the same thing, it is preferable to instead check whether count is less than 4.

```
3 while count < 4:
```

This test is exactly the same as the previous one—once count reaches 4, the loop will exit. But it has one added benefit: you can use the desired number of loop iterations in the boolean test—you don't need to do any math. If you want the loop to run exactly four times, you can use the boolean condition count < 4. If you want it to run a hundred times, you can use the condition count < 100. Remember, though, that you need to initialize count to 0, *not* 1, for this strategy to work as intended. Programmers tend to like counting from 0 rather than 1 for reasons we will see in the next chapter.

You can achieve the same effect, though, by starting count at 1 and using the boolean condition

```
3 while count <= 4:
```

By using the <= operator, you compensate for the fact that you are starting at 1 rather than 0. This configuration works the same way as the other one—the number in the loop's boolean test is number of loop iterations you want. The downside is that count no longer keeps track of the number of times you've printed 'hello'—it will actually be off by one.

6.4 Tabulating Legal Fees

Let's consider a program that allows an attorney to calculate the bill that she wishes to charge to a client. The attorney will enter the name of the client and billing rate per hour. The program will then ask the attorney to enter as many time intervals as she likes. When she enters a blank time interval, it will signal to the program that she is done and wishes to see a final bill. Here is a sample interaction.

```
Client name: Larry
Hourly billing rate: $200
Minutes spent: 10
Minutes spent: 60
Minutes spent: 87
Minutes spent:
Total bill for Larry: $523.3333333333334
```

To begin, our program will need to ask the user for the initial inputs.

```
1 # Ask for the client's name.
2 client = input('Client name: ')
3
4 # Ask for the billing rate.
5 rate = int(input('Hourly billing rate: $'))
```

We should also create a variable that will keep track of the total number of minutes the attorney has entered so far.

```
6 # Number of minutes the attorney has entered.
7 minutes = 0
```

Next, we will need a loop. For now, we will leave the loop's boolean condition blank—we can fill that in later.

```
8  # Collect billing information.
9  while ???:
10     additional_minutes = input('Minutes spent: ')
11
12     # If the user didn't leave the entry blank.
13     if additional_minutes != '':
14         minutes += int(additional_minutes)
```

On line 10, the program asks for a time interval to be billed. When this field is not left blank, the if-statement on line 13-14 converts this input into an integer and adds it to the total number of minutes entered so far (line 14).

Finally, after the loop ends, the program calculates and prints the output.

```
15 hours = minutes / 60
16 total = rate * hours
17 print('Total bill for ' + client + ': ' + str(total))
```

It converts the number of minutes worked into the number of hours worked and multiplies it by the attorney's hourly rate to determine the final bill, which it prints.

The last remaining piece of this program is figuring out the boolean condition for the while-loop. For this program in particular, picking the right boolean condition is no easy task. It will involve considering the tradeoffs of strategies that affect the code's elegance and readability in different ways. In the sections that follow, we will examine several approaches for filling in the boolean condition. This exploration will arm you with a number of different techniques for designing loops in the future.

6.4.1 Dummy Values

We know that the program should exit the loop when the variable additional_minutes is equal to the empty string (''), meaning that the attorney left her response to the input statement blank. Based on that information, we could try the following condition:

```
9  while additional_minutes != '':
```

or, equivalently

```
9  while len(additional_minutes) > 0:
```

Both of these tests ensure the same thing—that additional_minutes isn't empty. The first directly checks whether additional_minutes is the empty string, while the second checks whether the string that additional_minutes stores is more than zero characters long.

Neither is the right boolean condition, however, because, on Python's first pass through the loop, additional_minutes hasn't yet been defined. It doesn't

get defined until the first time the input statement on line 10 executes. Python will complain accordingly, and the program will crash.

This is the key challenge in designing the right exit condition for the loop—the program always has to ask for input at least once, so we need to somehow coax Python into executing the body of the loop once no matter what.

To fix this problem, we could define additional_minutes before the loop. But what value do we give it?

```
9  additional_minutes = ???
10 while additional_minutes != '':
11     additional_minutes = input('Minutes spent: ')
```

If we were to set it to the empty string, which is generally a reasonable default value for a string (much like setting an integer to 0), the loop's boolean guard would be False and the program would never enter the loop. Instead, we should set it to anything *except* the empty string. Below, we choose an arbitrary dummy value.

```
9  additional_minutes = 'dummy value'
10 while additional_minutes != '':
11     additional_minutes = input('Minutes spent: ')
```

This way, the program will enter the loop the first time through. The moment it does so, the dummy value of additional_minutes will be overwritten by the user's input, meaning it won't have any impact on the program's calculations. This approach is one way of solving the problem, but the use of an arbitrary dummy value seems like bad style and will probably confuse later readers of our code.

Rather than using a dummy value, we can take advantage of the special Python value None. You will recall that None represents the absence of information. It is specifically designed for situations where we need a dummy value. Rather than setting additional_minutes to an arbitrary string, we can set it to None.

```
9  additional_minutes = None
10 while additional_minutes is None or additional_minutes != '':
11     additional_minutes = input('Minutes spent: ')
```

In order to use None in this way, we had to modify the while loop's boolean test slightly. The test has two separate conditions. We will loop either if:

1. additional_minutes stores the value None, represented by the boolean expression additional_minutes is None.

2. additional_minutes contains the empty string, meaning the attorney is done entering time increments. Doing so uses the same boolean expression as before: additional_minutes != ''.

On the first iteration of the loop, additional_minutes will be None, ensuring the loop's boolean guard is True. On subsequent iterations, additional_minutes will

store a string that is not empty, which satisfies the second part of the loop's boolean guard. When additional_minutes is finally empty, neither boolean condition will hold—additional_minutes will be neither None nor a string that is non-empty—causing the loop to exit.

6.4.2 Repeating Code

We can avoid using a dummy value by rearranging the contents of the loop slightly.

```
9  additional_minutes = input('Minutes spent: ')
10 while additional_minutes != '':
11     minutes += int(additional_minutes)
12     additional_minutes = input('Minutes spent: ')
```

Notice how we ask for input twice—once before the loop and once at the very end. If the attorney types in a blank entry at the first request for input, the loop will never execute. If the attorney does type in a number, Python will enter the loop (line 11), add it to the running total (line 12), and ask for input again (line 13). Python then returns to line 10 and uses that input to determine whether to keep iterating through the loop.

This structure may initially seem counterintuitive. One way of understanding the code above is that we've shifted where the while-loop's boolean test takes place. Usually, the while-loop tests its boolean condition at the beginning of the loop—before its body executes. In this case, we want to ensure that the input statement executes at least once—that is, we want the body of the loop to execute at least once. One way of achieving this goal would be for the while-loop to test its boolean condition at the end of the loop's body rather than at the beginning. Unfortunately, Python doesn't provide a way to do this.

The code above does the next best thing. We've copied and pasted the loop body so that it appears once before the loop. This mimics the effect of putting the while-loop test at the end of the loop, since the body of the loop executes once on line 9 before the while loop test runs on line 10.

One benefit of this approach is that it eliminates the need for an if-statement in the loop, since the loop's boolean condition takes care of the same job. The primary shortcomings of this approach are that we have to repeat a line of code verbatim (which is usually a sign of bad style) and that a reader will probably be puzzled by the code at first glance. That said, this technique is a perfectly reasonable approach that is very common in practice. A descriptive comment can help to make our strategy clear to readers.

6.4.3 Boolean Variables

One alternative is to use a boolean variable to indicate whether it is our first time through the loop.

```
9  first_time = True
10 while first_time == True or additional_minutes != '':
```

```
11      first_time = False
```

Before the loop, we set the variable first_time to True. The loop's boolean condition has two parts. In order to enter the loop, either first_time must be True or additional_minutes must contain some text. On the first iteration, we rely on the fact that first_time will be True, since that's how we initialized it on the line before. But upon entering the loop, first_time is immediately set to False, so the only way the program will continue to loop is if additional_minutes is nonempty.

If you are paying very careful attention, you'll notice an apparent problem with this boolean test: on the first time through the loop, additional_minutes hasn't yet been defined, so won't the program crash as it did before? It turns out that it won't.

To speed up the execution of your program, Python short-circuits the boolean or and and operators. If the expression on the left of the or operator is True, Python doesn't even bother to evaluate the expression on the right, since the overall expression will evaluate to True no matter what. Python does the same with the and operator when the expression on the left is False. This means that the first time Python tests the boolean condition on line 10 of the code fragment above, it will see that the value of first_time is True and ignore the reference to the as-of-yet undefined variable additional_minutes. If we had written the condition the other way around:

```
additional_minutes != '' or first_time == True
```

Python would crash, complaining that additional_minutes is not defined.

One other style note: whenever you find yourself testing whether a boolean variable is True or False, you do not need to write

```
first_time == True or additional_minutes != ''
```

as we have done above. Instead, since the variable is already of type boolean, it is better to simply use the variable on its own:

```
first_time or additional_minutes != ''
```

To test whether a variable is False, the expression

```
first_time == False or additional_minutes != ''
```

could be simplified to

```
not(first_time) or additional_minutes != ''
```

To emphasize: when you are using a boolean variable (i.e., x) in the guard of an if-statement or a while-loop, do not test whether the variable x == True or x == False. Instead, just use x if testing for True and not(x) if testing for False.

Back to our billing program. We will consider one final strategy that relies on a boolean variable. Rather than keeping track of whether it is our first time through the loop, we could use a boolean variable to control whether the program should keep looping.

```
9  keepLooping = True
10 while keepLooping:
```

The variable keepLooping is initially set to True and has sole control over whether we continue to loop. If we never modify the variable inside the while-loop, the program will run forever. When should we change the variable? When the attorney types in a blank entry.

```
9  keepLooping = True
10 while keepLooping:
11     additional_minutes = input('Minutes spent: ')
12
13     # If the user didn't leave the entry blank.
14     if additional_minutes != '':
15         minutes += int(additional_minutes)
16     else:
17         keepLooping = False
```

Notice the change on lines 16 and 17. If the user types in a blank entry, the variable keepLooping is flipped from True to False, causing the loop to exit on the next time through.

6.4.4 Break

Using the break command, we can modify the previous program by abandoning the loop's boolean condition entirely.

```
9  while True:
10     additional_minutes = input('Minutes spent: ')
11
12     # If the user didn't leave the entry blank.
13     if additional_minutes != '':
14         minutes += int(additional_minutes)
15     else:
16         break
```

On line 9, the loop's boolean condition has been set to True, creating an infinite loop that will never exit without the help of break. The else-statement on lines 16 and 17 serves this purpose, exiting the loop if the attorney has made a blank entry.

This strategy is notably heavy-handed, but it gets to the heart of the issue that has made this loop so tricky to configure. We want to perform the loop's boolean test at the end of the loop's body rather than at the beginning. In other words, the loop's body should always execute once before we ever consider exiting.

While other languages have a special kind of loop designed specifically for this situation, Python does not, forcing us to grapple with a long list of alternative configurations. By placing the if-statement and break at the very end of

the loop while eliminating the loop's normal boolean test at the beginning, we have directly recreated this functionality.

6.4.5 Summary

None of the strategies we have described in the preceding pages is perfect, and it is up to you as a programmer to use your best judgement to create programs that are straightforward, readable, and—most importantly—correct. Below, we have reproduced the entire program from this section with the "repeating code" strategy.

```
1  # Ask for the client's name and billing rate
2  client = input('Client name: ')
3  rate = int(input('Hourly billing rate: $'))
4
5  # Number of minutes the attorney has entered.
6  minutes = 0
7
8  # Collect billing information.
9  additional_minutes = input('Minutes spent: ')
10 while additional_minutes != '':
11     minutes += int(additional_minutes)
12     additional_minutes = input('Minutes spent: ')
13
14 # Calculate the final bill.
15 hours = minutes / 60
16 total = rate * hours
17 print('Total bill for ' + client + ': ' + str(total))
```

6.5 Nested Loops

Suppose we wanted to extend the program from the previous section. Instead of handling just one client, the attorney wants the program to do her billing for all of her clients. She should be able to enter a client's name, calculate a bill, and then enter another client's name and do the same thing again. The program should stop only when she leaves the client's name blank.

```
Hourly billing rate: $200
Client name: Larry
Minutes spent: 6
Minutes spent: 60
Minutes spent:
Total bill for Larry: $220
Client name: Mary
Minutes spent: 30
Minutes spent:
```

```
Total bill for Mary: $100
Client name: Gary
Minutes spent: 27
Minutes spent: 18
Minutes spent:
Total bill for Gary: $150
Client name:
```

First, we need to ask for the hourly billing rate, just like we did before.

```
1  # Ask for the billing rate
2  rate = int(input('Hourly billing rate: $'))
```

Next, we will need to request each client from the attorney. Since this task is repeated, we should put it within a loop. We also know that this loop should exit if the attorney leaves the client's name blank. We could employ any one of the strategies discussed in the preceding pages. Here, we ask for the client's name both before the loop and again at the end of the loop.

```
3  # Loop through each client.
4  client = input('Client name: ')
5  while client != '':
6      # The rest of the program goes here.
7
8      # Request the name of the next client.
9      client = input('Client name: ')
```

The body of the while-loop should then repeatedly ask for time intervals, add them together, and calculate an overall bill. This is exactly the program we just wrote! We can copy and paste it, largely without modification, into the loop. In other words, we will be putting a loop within a loop. Just like when we nested if-statements, we need to make sure the nested loop is properly indented.

```
1  # Ask for the billing rate
2  rate = int(input('Hourly billing rate: $'))
3
4  # Loop through each client.
5  client = input('Client name: ')
6  while client != '':
7      # Number of minutes the attorney has entered.
8      minutes = 0
9
10     # Collect billing information.
11     additional_minutes = input('Minutes spent: ')
12     while additional_minutes != '':
13         minutes += int(additional_minutes)
14         additional_minutes = input('Minutes spent: ')
15
16     # Calculate the final bill.
```

```
17      hours = minutes / 60
18      total = rate * hours
19      print('Total bill for ' + client + ': ' + str(total))
20
21      # Request the name of the next client.
22      client = input('Client name: ')
```

We now have a complete program that implements the attorney's request. Line 2 asks for the attorney's billing rate. Lines 5 and 6 initiate the outer loop, which spends each iteration generating a bill for a new client. The body of this loop creates a single bill. It first sets the total number of minutes seen for the client to 0 (line 8). On lines 10 to 14, the program repeatedly requests blocks of time during which the attorney has worked for the client. It does so in the form of a nested while loop that has its own body on lines 13 to 14. When the attorney enters a blank time interval, meaning that the bill is complete, the nested loop exits and the program proceeds to lines 16 through 20 where the bill is calculated and printed. At the end of the body of the outer loop, the program asks for the name of the next client to be billed (lines 21 and 22) and the outer loop repeats again from line 6. If the client name was blank, the boolean condition on line 6 fails, causing the outer loop to exit. Since there is no code after the loop, the program also terminates.

6.6 Conclusion

In this chapter, we discussed loops, which have given you the ability to write programs that run for an unspecified (or even infinite) amount of time. Loops are fundamental to programming. In fact, theoretical computer scientists have proved that any programming language that has both loops and lists (the subject of the next chapter) is powerful enough to express any computation that could ever be written using any known programming language on any known computer (a concept called *Turing-completeness*).

The programs we have written in this chapter are long—on the order of 20 or more lines. They are also capable of accomplishing real, useful work. Your ability write longer programs that accomplish more sophisticated tasks will only continue to grow rapidly as you add to your arsenal of Python skills in the coming chapters. Now that you have equipped yourself with the two fundamental control-flow techniques—conditionals and while-loops—you will enhance your ability to manage and manipulate complicated information with several new types of data that extend the power of those you already know.

6.7 Cheatsheet

While-Loops

(Note that the phrases within angle brackets (< and >) should be substituted for actual Python code.)

```
1 while <boolean expression>:
2     <line of code within if-statement>
3     <optionally, another line of code within if-statement>
4
5 <line of code after if-statement>
6 <another line of code after if-statement>
```

A while-loop has four components.

1. It begins with the keyword while.
2. A boolean expression (a guard).
3. A colon.
4. One or more lines of code indented to the same level (a code block).

When Python encounters an while-loop (line 1), it evaluates the boolean expression. If the boolean expression evaluates to False, Python skips the indented code block and picks up at the next line indented to the same level as the while operator. (line 5). If the boolean expression evalutes to True, Python executes the indented code block (lines 2-3); afterwards, it returns to the while statement (line 1), re-runs the boolean expression, and repeats this process.

In other words, a while-loop is an if-statement that keeps executing its body as long as its boolean test continues to be True after each iteration.

While-loops can be nested in the code blocks of other expressions (like if-statements and other while-loops) and can themselves contain nested expressions.

Break and Continue

When it appears on its own line inside a loop, the break keyword causes Python to immediately exit the loop and continue to the next line of code afterwards (i.e., the next line of code indented to the same level as the while keyword). If it is within multiple nested loops break keyword exits only the single inner-most loop.

When it appears on its own line in a loop, the continue keyword causes Python to immediately return to the beginning of the loop and start the next iteration. If it is within multiple nested loops continue keyword only returns to the beginning of the single inner-most loop.

Common Mistakes

Useless Loop—A loop whose body will never run because its boolean condition will never be True on the first iteration.

Infinite Loop—A loop that will never exit because its boolean condition will never be False. It loops forever.

Off-by-one Error—A loop that runs for one too many times or for one too few times due to a counting error. Typically caused by mixing up the < and <= operators or initializing a counter variable to 0 instead of 1 (or vice versa).

Testing at the End of the Loop

In Python, while-loops test their boolean conditions at the beginning of each iteration. Sometimes, a program needs to ensure that the body of the while-loop evaluates at least once. It would be easy to do so if Python had a loop that tested its boolean condition at the end of each iteration rather than at the beginning, but Python has no such feature. Instead, there are several strategies to mimic this behavior (see Section 6.4). One strategy is to copy and paste the body of the loop before the loop begins:

```
1 <body of loop>
2 while <boolean condition>:
3     <body of loop again>
```

Keywords

Iteration—A single pass through a loop.

Loop—A block of code that repeats until a particular exit condition is met.

Short-Circuiting—Ignoring the expression to the right of the boolean and and or operators after evaluating the expression to the left because it is impossible for the expression on the right to change the result of evaluating the overall expression. Example: if the expression to the left of the or operator evaluates to True, the overall expression is guaranteed to evaluate to True, so Python ignores the expression to the right.

Chapter 7

Lists

In the previous few chapters of this book, we discussed the basics of composing programs in Python: receiving input, controlling the flow of a program, and computing output. So far, the data we have processed has been rather simple, comprising no more than a handful of numbers or words.

In this chapter, we will add a far more powerful mechanism for managing large amounts of data—lists. Lists allow a variable to store a "collection of strings" or a "collection of integers" rather than just one. Each item in a list is known as an element. A list could represent a set of clients (where each element is the client's name), a docket (where each element is a case to be heard), or even the US Code (where each element is a Title). Elements can be added to or removed from lists, making it possible to alter these lists afterward as new clients retain your services, a court modifies its schedule, or new laws are passed. Lists dramatically increase the expressive power of your programs, giving you the ability to store unlimited amounts of information.

The unifying theme of this chapter is managing large quantities of data. How do you tell Python to store 100,000 FCC comments or the majority opinions of every Supreme Court case from this past term? Lists.

Lists are the first new type of data we have introduced since Chapter 4, and they are different from the fundamental types in many ways. Most importantly, lists store other kinds of data, and the type of a list is determined by the kind of data that it stores. If a list stores integers, it is said to have the type "list of integers"; if it stores strings, it is a "list of strings."

As before, it is vital that you keep track of the type of data with which your code is working. If a statement expects a list of strings but receives a list of integers, it will either do something unexpected or crash when it tries to make use of the values that the list contains. It is equally important to clarify whether code expects an individual value or a list of values. Code that anticipates an integer is very likely to crash if it receives a list of integers instead.

7.1 Creating Lists

To explore the basics of lists, we will briefly return to interactive Python. To create a list, type an opening square bracket, several values separated by commas, and a closing square bracket.

```
>>> [1, 2, 3]
[1, 2, 3]
```

Recall that, when you type an expression in interactive Python, Python evaluates it and echoes the result back to you. Above, we have created a list that stores three values—the numbers 1, 2, and 3. We could have created a list with any number of values.

```
>>> [7, 4, 9, 5, 2]
[7, 4, 9, 5, 2]
>>> [8]
[8]
>>> [23, 22]
[23, 22]
```

Just as integers have the number 0 and strings have the empty string (`''`), you can also create the empty list, which doesn't contain any elements.

```
>>> []
[]
```

The empty list is useful for the same reasons that 0 and `''` are—they are good initial values for variables that will add together several integers or concatenate several strings. As we will see later, the empty list is a useful starting point for slowly building a list over time.

Lists can also store any type of value.

```
>>> ['hello', 'world']
['hello', 'world']
>>> [3.14, 2.79, .675]
[3.14, 2.79, .675]
>>> [True, False, False, True]
[True, False, False, True]
```

When creating a list, you can use expressions in addition to values.

```
>>> [3 * 5, 2 + 2, 3 ** 3]
[15, 4, 27]
```

Note that Python allows you to create lists of values with mixed types.

```
>>> [22, 'Thomas Jefferson', True]
[22, 'Thomas Jefferson', True]
```

However, this behavior is dangerous if not used carefully. It is generally better to create lists that store only one type whenever possible. When you do need

to create a list that mixes values of different types, make sure you are certain exactly which places in the list will have which types. For emphasis, we strongly discourage you to ever create lists that contain more than one type of values.

You can even store a list inside another list. That list, in turn, will contain other values. It might even contain another list.

```
>>> [[1, 2], [3, 4, 5], [6]]
[[1,  2],  [3,  4,  5],  [6]]
```

When dealing with nested lists, it is particularly important that you carefully keep track of types. The list above has the type "list of lists of integers"—every element that it stores has type "list of integers." As always, you should be consistent about the types of items you store in a list—the level of nesting should be the same for every element of the list. This nesting can go arbitrarily deep.

```
>>> [[[2, 3], [4]], [], [[6]]]
[[[2,  3],  [4]],  []  [[6]]]
```

The list above is a "list of lists of lists of integers." The first element of the list is `[[2, 3], [4]]`, which is a "list of lists of integers". In turn, this nested list has two elements, `[2, 3]` and `[4]`, each of which are lists of integers. The second element of the main list is `[]`, the empty list. The third element is `[[6]]`, a one-element list of lists of integers that contains the list `[6]`. Although nested lists may seem confusing and a little contrived, they are extraordinarily useful in practice. What if you wanted to store the students in the 1L class separated by section? Each section could be represented by a list of names (as strings), and these lists could be stored together within one larger list.

Be careful when a list contains the empty list as one of its elements. This is perfectly legal but a little confusing. The list `[[]]` is not empty—it is a list of lists with one element, and that element just happens to be the empty list.

Finally just like any other value, lists can be saved to variables.

```
>>> evens = [2, 4, 6]
>>> evens
[2,  4,  6]
```

7.2 List Length

You may recall that the len function evaluated to the number of characters in its string argument.

```
>>> len('hello')
5
>>> len('constitution')
12
>>> len('')
0
```

The len function works identically for lists. Provided with a list as an argument, it evaluates to the number of elements in the list.

```
>>> len([7, 11, 15])
3
>>> len(['we', 'hold', 'these', 'truths'])
4
>>> len([])
0
```

It is no mere coincidence that the len function works on both lists and strings. As it turns out, strings are just another kind of list. We will return to this topic in more detail in the chapter on strings.

If you pass len a list of lists with nested elements, the len function will count the number of elements in only the outer list. Concretely, the list `[[1, 2, 3, 4]]`, a list containing a list of the numbers 1 through 4, has length 1. The outer list is a list of lists of integers containing a single element, the list `[1, 2, 3, 4]`. Similarly, the list `[['to', 'be'], ['self', 'evident']]` has length 2, since the outer list contains two elements: the lists `['to', 'be']` and `['self', 'evident']`. As a final example, consider the list `[[], [7], []]`. It has length 3, since the outer list contains three elements—the empty list twice and `[7]`.

7.3 Accessing List Elements

When it comes to lists, order matters. The list `[1, 2]` is different from the list `[2, 1]`. This ordering is what makes it possible to extract the individual elements contained in the list. You can access these elements via their numerical positions within the list (counting from left to right). Simply write the name of the variable that stores the list followed by square brackets containing the integer position of the value you want.

```
name_of_list[12]
```

Before showing a concrete example, note that Python starts counting at 0, not 1. To emphasize this point further, the first value in the list is at position 0 and the second value in the list is at position 1. There are historical peculiarities that explain this strange behavior, but the convention has stuck and nearly all computer languages start counting at 0.

```
>>> justices = ['Anthony', 'Clarence', 'Elena', 'John']
>>> justices[0]
'Anthony'
>>> justices[1]
'Clarence'
>>> justices[2]
'Elena'
>>> justices[3]
'John'
```

The integer between the square brackets is known as the index or subscript, and the process of accessing a value from a list is called indexing or subscripting. The expression `justices [2]` is often read in English as "justices index 2" or "justices sub 2."

What happens if you attempt to access an index beyond the end of the list?

```
>>> justices[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Predictably, the program crashes and Python produces a clear error. You tried to access an index that was "out of range"—beyond the end of the list. If you are ever in a situation where you aren't sure whether an index will be in range (such as when it is provided by a user as input), you should always test it with an if-statement that checks whether the index is smaller than the length of the list.

Notice that, since Python starts counting at 0, the length of a list is never a valid index. For example, the list `justices` has four elements, but using the index 4 would go off the end of the list. The highest index in a list is always one less than the length of the list. To retrieve the last element of a list, you would write the following.

```
>>> justices[len(justices) - 1]
'John'
```

If you try to supply the length of the list as an index, Python will crash.

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

Surprisingly, using a negative index will not necessarily cause Python to crash. Negative indices count backwards—from right to left or from highest index to lowest index—so list index -1 is the last element in the list.

```
>>> justices[-1]
'John'
>>> justices[-2]
'Elena'
>>> justices[-3]
'Clarence'
>>> justices[-4]
'Anthony'
```

Once these negative indices reach the first element in the list and run out of list elements, Python will produce a similar "out of range" error.

```
>>> justices[-5]
```

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

List indexing is one key reason why Python distinguishes integers from floating point numbers. Since it makes no sense to access the value at position 3.8 of a list, Python permits only indices that have type integer. Python will even reject floating point indices that don't have anything after the decimal point.

```
>>> justices[3.8]
Traceback (most recent call last):
  File "<stdin">, line 1, in <module>
TypeError: list indices must be integers or slices, not float
>>> justices[2.0]
Traceback (most recent call last):
  File "<stdin">, line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

Indices can, however, be any valid integer expression. This includes variables and mathematical expressions. Python will evaluate the expression within square brackets before indexing the list.

```
>>> i = 2
>>> justices[i]
'Elena'
>>> i = i - 1
>>> justices[i]
'Clarence'
>>> justices[5 * i - 2]
'John'
```

List indexing should explain the formerly mysterious command-line argument notation. You can access the command line arguments by first importing the sys library and then writing `sys.argv[1]` for the first argument, `sys.argv[2]` for the second argument, etc. In hindsight, it should be clear that `sys.argv` is a list of strings storing the command line arguments. But why do we starting counting at 1? The value of `sys.argv[0]` is the name of the program that was executed. The second item in the list (index 1) is the first command line argument.

You will need to index multiple times when you are accessing the elements of nested list. For each level in the list, you will need to provide a separate index. Consider the list below, which includes the justices on the Supreme Court organized by the president who appointed them.

```
>>> justices_by_prez = [['Kennedy'], ['Thomas'], ['Ginsburg', 'Breyer'],
    ['Roberts', 'Alito'], ['Sotomayor', 'Kagan']]
```

The justices appointed by Bill Clinton are in the third nested list.

```
>>> clinton = justices_by_prez[2]
>>> clinton
['Ginsburg', 'Breyer']
```

Notice that the index was used to return one of the interior, nested lists, and then a new variable was assigned to that list. To access Justice Ginsburg, you could then index the variable clinton.

```
>>> clinton[0]
'Ginsburg'
```

What if you wanted to access Justice Ginsburg's name without having to perform the intermediate step of extracting the Clinton appointees? You can combine these two indexing steps into one.

```
>>> justices_by_prez[2][0]
'Ginsburg'
```

Each index needs to be in separate square brackets since it is referring to a different list. The first index (2) extracts the Clinton appointees from the larger list. The second index (0) extracts Justice Ginsburg's name from the nested list of Clinton appointees.

7.4 Slicing

Sometimes, rather than extracting an individual element, you will want to obtain a list that contains some smaller portion of the original list. For example, if you wanted only the Clinton appointees to the Supreme Court, you would want

```
['Barack', 'Chester']
```

from the larger list

```
['Abe', 'Barack', 'Chester', 'Dwight']
```

You could individually extract the elements and build a new list.

```
>>> presidents = ['Abe', 'Barack', 'Chester', 'Dwight']
>>> middle = [presidents[1], presidents[2]]
>>> middle
['Barack', 'Chester']
```

The second command creates a new list by indexing presidents to extract 'Barack' and 'Chester', storing the resulting list in the variable middle.

Python provides a much more convenient way of doing the same job known as slicing.

```
>>> presidents[1:3]
['Barack', 'Chester']
```

Slicing a list looks similar to indexing one, but produces a new list derived from the original. Just as when indexing, type the name of the list followed by square brackets. Inside those square brackets, write two integers separated by a colon. These integers tell Python the list indices at which to start and stop slicing. The first integer is the starting index of the slice, and the second number is the

index *after* the last index of the slice. So in the code above, the first index is 1 (the index of 'Barack' in the original list) and the second is 3 (the index after the index of 'Chester').

Slicing doesn't modify the original list—it generates a new list with the appropriate elements copied from the original list.

If you want a slice that goes all the way to the end of the list, you will need to provide a second index that is one beyond the highest index in the list. For example, to obtain the final three names in presidents, the second index of the slice will need to be 4.

```
>>> presidents[1:4]
['Barack', 'Chester', 'Dwight']
```

This may seem uncomfortable at first, since doing normal indexing beyond the end of the list (i.e., president[4]) will cause Python to crash, whereas doing so with slices is often unavoidable.

Conveniently, you can simply leave off the second index when you intend for a slice to go all the way to the end of the list. Be careful not to forget the colon!

```
>>> president[1:]
['Barack', 'Chester', 'Dwight']
```

Likewise, you can leave off the first index if you intend for a slice to start at the very beginning of the list.

```
>>> presidents[:2]
['Abe', 'Barack']
```

Slices can include negative indices, which count from the right just as before.

```
>>> presidents[1:-1]
['Barack', 'Chester']
```

Since the starting index is 1, the slice begins with the second element of the list. It ends just before the last element of the list since -1 refers to the first position from the right.

Consider another example.

```
>>> presidents[-4:-2]
['Abe', 'Barack']
```

The index -4 refers to the fourth element from the right, which is the first element in the list, so the slice starts there. The index -2 refers to the second element from the right, which is the third position of the list. The slice stops one before this element, so it includes up through only the second position of the list.

When the slice operators don't include any elements (such as [3:3]) or when the starting index is greater than the ending index (such as [2:1]) the expression simply evaluates to the empty list.

```
>>> presidents[3:3]
```

```
[]
>>> presidents[2:1]
[]
```

When one of the indices goes beyond the end of the list to the left or right, Python will include the list all the way up to the leftmost or rightmost element. It won't generate an error.

```
>>> presidents[1:100]
['Barack', 'Chester', 'Dwight']
>>> presidents[-8:2]
['Abe', 'Barack']
```

7.5 Modifying Lists

Thus far, we have used list indices to access individual items or sublists (using the slicing notation). Sometimes, we have done this in interactive Python, which displays the result of doing so to the screen. In a few instances, we did this on the right side of the = operator, thus assigning the result to a variable on the left side.

Indices and slices can also appear on the left side of the = operator just like a variable. Doing so tells Python to modify the value stored at a particular index of the list. To do so, write a normal variable assignment statement but follow the name of a variable containing a list with the index of the element you want to overwrite in square brackets.

```
>>> presidents
['Abe', 'Barack','Chester', 'Dwight']
>>> presidents[1] = 'Benjamin'
>>> presidents
['Abe', 'Benjamin','Chester', 'Dwight']
```

You can also use slices to modify more than one value at a time.

```
>>> presidents[1:3] = ['Bill', 'Calvin']
>>> presidents
['Abe', 'Bill','Calvin', 'Dwight']
```

The items on the right side of the assignment replaced the items specified by the slice on the left. The number of items assigned to a slice need not equal the number of items in the slice, meaning slice assignment can grow or shrink the length of the list.

```
>>> presidents[1:3] =['Millard']
>>> presidents
['Abe', 'Millard', 'Dwight']
>>> presidents[1:3] = ['Lyndon', 'Herbert', 'Woodrow']
>>> presidents
['Abe', 'Lyndon', 'Herbert', 'Woodrow']
```

Finally, notice that the difference between using the slice operator and using a single index:

```
>>> presidents
['Abe', 'Lyndon', 'Herbert', 'Woodrow']
>>> presidents[1:3] =['Grover', 'Rutherford']
>>> presidents
['Abe', 'Grover', 'Rutherford', 'Woodrow']
>>> presidents[1] = '['Zachary', 'Martin']'
>>> presidents
['Abe', ['Zachary', 'Martin'], 'Rutherford', 'Woodrow']
```

When we put a slice to the left of the = operator and a list to the right, the elements of the list on the right are integrated into the list on the left. When we put an index to the left of the = operator and a list to the right, the index on the left is modified to store the list on the right. In the first instance, the result is a list of strings. In the second instance, the result is a list of strings whose second element is itself a nested list of strings. The difference is subtle but crucial.

7.6 Range

As you write more complicated Python programs, you will find yourself generating lists of numbers on a very frequent basis. Often, you want to create the list all integers between 0 and some bigger number n or between 1 and n (for example, all of the integers from 1 to 46). Python provides a function, range, that is specially designed for this purpose.

The simplest way of using the range function is to provide it with a single integer argument. It will produce a list of integers starting at 0 and stopping one number before the argument.

```
>>> range(5)
range(0, 5)
```

Note that the call to the range function did not evaluate to a list. In Python 3, range produces a value that is similar to but not exactly a list. The precise explanation is beyond the scope of this book (look up "python generators" if you're curious), so you need to remember the following trick: In order to turn the result of the range function into a list, you will need to use the list type-casting function.

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

If you want the list to start at a number other than 0, you can provide range with two integer arguments. The first argument is the number at which you want the range to start, and the second number is the upper bound as before.

```
>>> list(range(7, 11))
[7, 8, 9, 10]
>>> list(range(-4, 1))
[−4, −3, −2, −1, 0]
```

You can also control the amount that the range function increases or decreases by using a third argument. This is known as the "step" or "step size." By default, the range will always increase by a step of 1, but it's often useful to count by 3's or count downwards.

```
>>> list(range(1, 16, 3))
[1, 4, 7, 10, 13]
>>> list(range(8, 2, -1))
[8, 7, 6, 5, 4, 3]
```

When you provide a range that doesn't contain any numbers, for example when your range counts upwards but the starting number is higher than the ending number, the range function will simply produce the empty list.

```
>>> list(range(4, 2))
[]
```

Right now, range may seem like an esoteric curiosity. When you begin using lists in combination with loops, however, range will quickly become one of your most commonly used functions.

7.7 Manipulating Lists

Since lists are so fundamental to programming in Python, the language provides a wide range of commands and functions that make common operations more convenient. These tools can be divided into three categories: functions, operators, and methods.

7.7.1 Functions

You are already quite familiar with functions. Among the functions you have already encountered are print, input, and len. Recall that, to use a function, you type its name followed by parentheses enclosing any arguments you want to pass to it.

```
print('Hello, world!')
```

Doing so will call the function, causing it to evaluate and possibly calculate a result. The input function, for example, causes Python to prompt the user to enter text; when the user is done, the call to the input function evaluates to the string that the user typed in. We have already mentioned the len function, which takes a list as an argument and evaluates to the number of elements it contains.

```
>>> presidents = ['George', 'Tom', 'Abe', 'Teddy']
>>> len(presidents)
4
>>> len([2, 4, 6])
3
>>> len([])
0
```

Python also includes several other functions for analyzing lists. The min and max functions compute the biggest and smallest elements in a list.

```
>>> jenny = [8, 6, 7, 5, 3, 0, 9]
>>> min(jenny)
0
>>> max(jenny)
9
```

The min and max functions work on many other kinds of data, including strings. When calculating the "biggest" or "smallest" string, Python uses its built-in notion of alphabetical order. Letters that come earlier in the alphabet (`'c'`) are "smaller" than letters that come later (`'q'`). Uppercase letters are smaller than every lowercase letter, so `'Z'` is smaller than `'a'`.

```
>>> gburg = ['Four', 'score', 'and', 'seven', 'years', 'ago']
>>> min(gburg)
'Four'
>>> max(gburg)
'years'
```

There are some types of data for which Python simply can't calculate a minimum or a maximum. For example, it will complain if you try to mix types.

```
>>> min([1, 'a'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

For many other types of data, Python will calculate a minimum or maximum, but it will do so in cryptic and hard-to-predict ways. You should avoid these situations, since they make your code harder to read and increase the likelihood that you will make a mistake.

```
>>> max([[1, 2, 3], [4], [], [5, 6], [12]])
[12]
```

In the example above, could you have predicted that [12] would be the "biggest" element in the list of lists above?

Using the same orderings as the min and max functions, Python provides a way to sort lists with the sorted function.

```
>>> amtrak = ['WAS', 'BAL', 'WIL', 'PHL', 'TRE', 'NWK', 'NYP']
>>> sorted(amtrak)
['BAL', 'NWK', 'NYP', 'PHL', 'TRE', 'WAS', 'WIL']
```

Note that the sorted function doesn't modify the original list—it creates a new list that contains the same elements but is in sorted order.

```
>>> pi = [3, 1, 4, 1, 5, 9, 2]
>>> sorted_pi = sorted(pi)
>>> sorted_pi
[1, 1, 2, 3, 4, 5, 9]
>>> pi
[3, 1, 4, 1, 5, 9, 2]
```

Even after calling sorted on pi, pi itself remained unchanged.

7.7.2 Operators

You have also studied operators extensively throughout this book. Operators are mathematical symbols like $+$, $-$, and $*$ and boolean operators like and, or, and == that are written between the values that they operate on. For example, to add two numbers you would use the + operator by typing

```
2 + 3
```

Python includes several useful operators for lists, some of which you are already familiar with. Just as with strings, you can combine (concatenate) the contents of two lists with the + operator.

```
>>> preamble1 = ['We', 'the', 'people']
>>> preamble2 = ['of', 'the', 'United', 'States']
>>> preamble1 + preamble2
['We', 'the', 'people', 'of', 'the', 'United', 'States']
>>> preamble2 + preamble1
['of', 'the', 'United', 'States', 'We', 'the', people']
>>> preamble1 + preamble1 + preamble1
['We', 'the', people', 'We', 'the', people', 'We', 'the', people']
```

Notice that the + operator merges the two lists into one "flat" list. It doesn't create nested lists. Again like strings, you can use the $*$ operator and an integer to replicate a list, repeat its contents the specified number of times. You can put the integer and the list in either order.

```
>>> primes = [2, 3, 5, 7]
>>> primes * 3
[2, 3, 5, 7, 2, 3, 5, 7, 2, 3, 5, 7]
>>> 2 * primes
[2, 3, 5, 7, 2, 3, 5, 7]
```

Python also provides two boolean operators for lists that allow you to check whether a list contains a particular element. You can use the in operator to

ask whether an element (say, 7) is in a list ([2, 4, 6, 8]). The expression will evaluate to True if the element is present and False otherwise.

```
>>> 7 in [2, 4, 6, 8]
False
>>> 7 in [1, 3, 5, 7]
True
```

As with all boolean operators, the in operator is particularly potent in combination with if-statements.

```
1  # Airports that our airline serves.
2  airports = ['dca', 'bwi', 'iad', 'jfk', 'lga', 'ewr']
3
4  # Collect airport information.
5  print('Welcome to the airline booking system.')
6  departure = input('Departure airport: ')
7  arrival = input('Arrival airport: ')
8
9  # Ensure these airports exist.
10 if (departure in airports) and (arrival in airports):
11     print('We fly to both of these airports!')
12     # Continue with booking...
13 else:
14     print('We do not fly to one or more of these airports.')
```

On line 2, the list of airports that the airline serves are stored in a list. The user is asked to provide desired departure and arrival airports on lines 6 and 7. On line 10, the program checks to ensure that these airports are actually present in the list using the in operator. If they are, the program will continue with the booking (line 12); if not, it will print a message to this effect (line 14).

```
Welcome to the airline booking system.
Departure airport: bwi
Arrival airport: ewr
We fly to both of these airports!

Welcome to the airline booking system.
Departure airport: dca
Arrival airport: lax
We do not fly to one or more of these airports.
```

Complementing the in operator is the not in operator, which checks whether an element *isn't* in a list. Writing

```
7 not in [2, 4, 6, 8]
```

will evaluate to False if 7 is in the list and False otherwise.

```
>>> 7 not in [2, 4, 6, 8]
```

```
True
>>> 7 not in [1, 3, 5, 7]
False
```

Finally, the del operator allows you to delete an element of the list based on its index.

```
>>> odds = [1, 3, 5, 7]
>>> del odds[2]
>>> odds
[1, 3, 7]
>>> del odds[0]
>>> odds
[3, 7]
```

The del operator deletes the element at the index you specify and shifts the indices of all of the items in the list that come afterward to fill in the gap.

You can even delete entire slices of a list.

```
>>> evens = [2, 4, 6, 8]
>>> del evens[1:3]
>>> evens
[2, 8]
```

7.7.3 Methods

Methods are a new way of interacting with data in Python, so we will spend slightly more time discussing how they work. Methods are an integral part of many of the libraries we will use to search large repositories of text or crawl the web, so it is critical to understand how methods work in some depth.

A method is identical to a function—it is called by typing its name followed by zero or more arguments separated by commas and enclosed in parentheses. The only difference is one small change in notation. Whereas you sort the names of Supreme Court justices using the sorted function by writing

```
sorted( justices )
```

Python includes a similar method that is simply called sort and is written

```
justices .sort ()
```

To call a method on a variable, write the name of the variable followed by a dot, the name of the method, and parentheses containing any arguments. For all intents and purposes, a method operates in exactly the same way as a function; you can think of the variable to the left of the dot as just another argument. The variable to the left of the dot as known as the method's object. In the example above, justices is the object of the call to the sort method. This terminology relates to a much bigger topic called object-oriented programming; this topic is beyond the scope of this book, but we will see evidence of its presence throughout the next few chapters.

Why does Python include both function *and* methods when they are effectively the same? And why are some purposes accomplished with methods and some with functions? By convention, methods and functions tend to be used in slightly different ways (even though Python does not force anyone to follow this convention).

For an explanation by example, let's take a closer look at the sorted function and the sort method. Both are provided a single list, the sorted function as its argument and the sort method as its object. As we saw before, the sorted function does not modify its argument.

```
>>> greek = ['epsilon', 'beta', 'gamma', 'delta', 'alpha']
>>> sorted(greek)
['alpha', 'beta', 'delta', 'epsilon', 'gamma']
>>> greek
['epsilon', 'beta', 'gamma', 'delta', 'alpha']
```

Instead, the sorted function evaluates to a new list that has the same elements as greek but is in sorted order. If we so desired, we could save this new list to a variable.

The sort method behaves a little differently.

```
>>> greek = ['epsilon', 'beta', 'gamma', 'delta', 'alpha']
>>> greek.sort()
>>> greek
['alpha', 'beta', 'delta', 'epsilon', 'gamma']
```

Notice the difference after the second command is typed. The call to the sort method doesn't evaluate to a result. Where the sorted function evaluated to a sorted copy of the list, the sort method modifies the list itself (as you can see after the third command). The actual call to the sort method evaluates to the value None—nothing. This distinction is subtle but vitally important. At least for lists, but also for other data types, this tends to be the difference between methods and functions. Although this convention is not a universal rule, functions and operators tend to create and evaluate to new values (new integers, new lists, etc.) without modifying their arguments. Methods tend to modify their objects and evaluate to None.

For another example, consider the difference between the + operator for lists (which, you'll recall, concatenates two lists together) and the extend method. Below, we concatenate two lists with the + operator.

```
>>> al_east1 = ['Yankees', 'Blue Jays']
>>> al_east2 = ['Orioles', 'Red Sox', 'Rays']
>>> al_east1 + al_east2
['Yankees', 'Blue Jays', 'Orioles', 'Red Sox', 'Rays']
>>> al_east1
['Yankees', 'Blue Jays']
>>> al_east2
['Orioles', 'Red Sox', 'Rays']
```

The + operator evaluated to a new list that contained the contents of al_east1 and al_east2 concatenated together. Neither of the original lists were modified.

In contrast, the extend method has a list as its object and takes a second list as an argument. It modifies its object, concatenating the elements of its argument.

```
>>> al_east1 = ['Yankees', 'Blue Jays']
>>> al_east2 = ['Orioles', 'Red Sox', 'Rays']
>>> al_east1.extend(al_east2)
>>> al_east1
['Yankees', 'Blue Jays', 'Orioles', 'Red Sox', 'Rays']
>>> al_east2
['Orioles', 'Red Sox', 'Rays']
```

The argument of the extend method (al_east2) was not modified, but the object was. Again, note that this distinction between methods and objects is by convention, not by law, but most parts of Python tend to follow it closely.

The object of a method (as the expression to the left of the dot is called) need not just be a variable. You can put any expression you want, enclosed in parentheses if necessary, to the left of the dot.

```
>>> ['Nationals', 'Marlins', 'Mets', 'Phillies'].sort()
```

The problem with using a method on something other than a variable is that, if the method modifies its object and evaluates to None, there is no way to retrieve the result. In the example above, we have no way to access the sorted list, since we never saved it to a variable.

Many of the most useful ways of interacting with lists are via methods. You can reverse the order of a list's elements, turning the front of the list into the back and vice versa, with the reverse method. Like sort, the reverse method does not need any arguments.

```
>>> stooges = ['Moe', 'Curly', 'Larry']
>>> stooges.reverse()
>>> stooges
['Larry', 'Curly', 'Moe']
```

You can add a single item to the end of a list with the append method.

```
>>> stooges = ['Moe', 'Curly']
>>> stooges.append('Larry')
>>> stooges
['Moe', 'Curly', 'Larry']
```

You can discard the last item using the pop method.

```
>>> stooges = ['Moe', 'Curly', 'Larry']
>>> stooges.pop()
'Larry'
>>> stooges
['Moe', 'Curly']
```

The pop method did two things. First, it removed the last item from the list stooges. But it also evaluated to the last item (the second command). This is the first time we have encountered a method evaluated to a value, but this is quite common in practice. If we wanted, we could capture that value in a variable and use it later.

```
>>> stooges = ['Moe', 'Curly', 'Larry']
>>> last_stooge = stooges.pop()
>>> last_stooge
'Larry'
>>> stooges
['Moe', 'Curly']
```

You can also provide the pop method with an integer argument specifying the index of the item that you want to access and remove.

```
>>> stooges = ['Moe', 'Curly', 'Larry']
>>> stooge = stooges.pop(1)
'Curly'
>>> stooges
['Moe', 'Larry']
```

Similarly, you can add an element to a list at an index of your choosing using the insert method. The method takes two arguments: the index at which to add the element and the element itself.

```
>>> stooges = ['Moe', 'Curly', 'Larry']
>>> stooges.insert(2, 'Shemp')
>>> stooges
['Moe', 'Curly', 'Shemp', 'Larry']
```

The indices of the existing elements of the list are shifted to make room for the addition.

There are several list methods that work by searching for a particular element rather than by specifying an index. You can find the index of an item in the list using the index method.

```
>>> stooges = ['Moe', 'Curly', 'Larry']
>>> stooges.index('Moe')
0
```

The index method doesn't modify the list in any way. If the element you're searching for isn't found, however, Python will crash.

```
>>> stooges = ['Moe', 'Curly', 'Larry']
>>> stooges.index('Shemp')
Traceback (most recent call last):
  File "<stdin", line 1, in <module>
ValueError: 'Shemp' is not in list
```

You can calculate the number of times that an item appears in a list with the count method.

```
>>> pi = [3, 1, 4, 1, 5, 9, 2, 6, 5]
>>> pi.count(5)
2
>>> pi.count(4)
1
>>> pi.count(7)
0
```

You can remove the first appearance of an element (starting from the left) with the remove method.

```
>>> pi = [3, 1, 4, 1, 5, 9, 2]
>>> pi.remove(1)
>>> pi
[3, 4, 1, 5, 9, 2]
>>> pi.remove(1)
>>> pi
[3, 4, 5, 9, 2]
```

To emphasize, the remove method removes only the first appearance of an element. You will need to repeatedly call remove if the element appears more than once. If you try to remove an element that isn't present in the list, Python will produce an error.

```
>>> pi.remove(7)
Traceback (most recent call last):
  File "<stdin", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Notice that all of the methods we have just described operate specifically on lists. The methods available on the right side of a dot depend on the type of the object on the left side of the dot. Once again, types matter in Python.

You will likely find it useful to protect calls to the index or remove methods with if-statements. For example, to ensure that an element is in the list before trying to remove it, you might write

```
1 pi = [3, 1, 4, 1, 5, 9, 2]
2 if 1 in pi:
3     pi.remove(1)
```

By changing the keyword if to while, the code fragment will remove *all* instances of an element from the list.

```
1 pi = [3, 1, 4, 1, 5, 9, 2]
2 while 1 in pi:
3     pi.remove(1)
```

The while-loop test on line 2 will continue to be True so long as the element 1 is in the list. Only once all instances of the element have been removed will the loop exit. If the element wasn't in the list at all, the boolean test would fail from the start and the loop would never execute.

7.8 Reference Types

In Chapter 5, when we introduced variables, we investigated a simple and seemingly innocuous question whose answer will change substantially once lists are involved. Suppose that we have a variable, x.

```
>>> x = 1
```

We can visually portray this assignment by drawing a box that represents the storage space on your computer that is reserved for x

We then create a new variable, y.

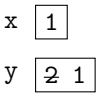
```
>>> y = 2
```

Python will give y its own storage space and set it to the value 2.

What if we then set y equal to x?

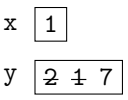
```
>>> y = x
```

Here is what our visual diagram looks like now.



Since the two variables store their values in separate storage areas, the value in x's box is copied to y's box. We can then modify y without impacting x.

```
>>> y = 7
>>> x
1
```



Since setting y equal to x caused Python to copy x's value to y while keeping their respective storage spaces separate, we say that x was copied by value. So far, this discussion should be review. We explored the same phenomenon in Chapter 5, and many of our examples have been predicated on the notion that Python behaves in this fashion.

Lists, however, work differently. Suppose we create a list, a.

```
>>> a = [1, 2]
```

We also create another list, b.

```
>>> b = [3, 4]
```

So far, this example is proceeding just like the last one, except with lists rather than integers. To continue following the pattern, we set b equal to a.

```
>>> b = a
```

Just as we expected, b should now have the same value that a had before.

```
>>> b
[1, 2]
```

Here is where things change. What happens if we modify b?

```
>>> b[0] = 12
```

The value of b changes as we expect.

```
>>> b
[12, 2]
```

But so does the value of a!

```
>>> a
[12, 2]
```

Setting b equal to a has somehow linked the two variables' values together.

```
>>> a
[12, 2]
>>> b
[12, 2]
>>> a.extend([3, 4, 5])
>>> a
[12, 2, 3, 4, 5]
>>> b
[12, 2, 3, 4, 5]
```

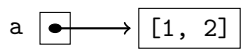
Why do lists behave differently than integers? Python integers, floating point numbers, and booleans are copied by value. This means that moving a value from one variable to another creates two separate copies. In contrast, lists (and nearly every other Python data type you will encounter) are copied by reference. Copying by reference means that, when a value is moved from one variable to another, the variables share one copy of the value. Changing one variable changes the shared copy, thereby changing both variables.

This concept is easier to understand visually. When we create an integer variable x, Python gives the variable storage space for its value.

```
>>> x = 1
```

Creating a list looks different.

```
>>> a = [1, 2]
```



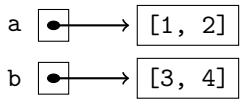
When we create the list [1, 2], Python sets aside storage space just for the list, independent of any variable. This is the box that surrounds the list on the far right of the diagram. When we save this list to a variable, Python

creates a reference from the variable's storage space to the list's storage space (represented by the arrow).

There are a number of reasons why Python gives a list its own storage space but doesn't do the same for an integer. The most important explanation is that lists can vary enormously in size—a list can be empty or it can contain a million elements. Its size can also vary over the course of its lifetime as elements are added or removed. Python can achieve this flexibility much more easily when the list has its own, separate storage space. Numbers, in contrast, always take up the same amount of space no matter how big or small they may be.

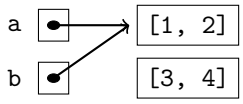
When we assign the second list to b, Python gives the list its own storage space and creates another reference.

```
>>> b = [3, 4]
```



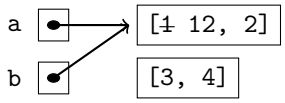
Finally, when we set b to a, a's *reference* is copied to b.

```
>>> b = a
```



This means that a and b now have the same reference—they point to the same storage space and share the same list. Modifying b's list will modify a's list. (In the process, the list that b used to point to, [3, 4], has been orphaned because it is completely inaccessible from any variable. Python will recycle the space that it takes up.)

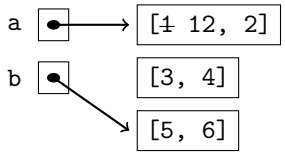
```
>>> b[0] = 12
```



By changing the first element of b, we have also changed the first element of a.

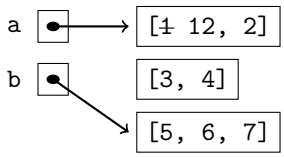
If we change b's reference itself, however, a is unaffected. For example, if we assign b a new list, b's reference will no longer point to the same reference as a and the two variables will no longer impact one another.

```
>>> b = [5, 6]
```



Python creates new storage space for the new list and modifies b's reference to point to it. a's reference is unchanged and continues to point to the original list. Altering the list that b refers to will not impact a.

```
>>> b.append(7)
```



The difference between copying by value and copying by reference may seem esoteric at first—at the moment, you may believe that it is far too low-level for a book that purports to teach programming to lawyers. We admit that most practical situations won't require you to understand the difference between the two paradigms. With that said, you will consider the time you invested reading the past few pages extraordinarily well spent when you encounter a bug whose solution rests on understanding values and references—a common occurrence that would have taken you hours to unravel without this knowledge.

Looking down the road, the difference between copying by value and copying by reference will resurface as a central theme when we discuss functions in Chapter 11 and will remain a quiet refrain throughout the rest of the book.

7.9 Tuples

Python includes several other list-like types—forms of data that store multiple elements and make them accessible based on their positions from front to back. We will spend the entirety of the next chapter discussing one of the most important of these list-like types, strings. Here, we will briefly introduce one other list-like type—tuples—whose distinctions from normal lists are so subtle that you will often forget you are working with a different type of data.

Tuples are created just like lists, except that they are surrounded with parentheses rather than square brackets.

```
>>> pres_list = ['Washington', 'Adams', 'Jefferson']
>>> pres_tuple = ('Washington', 'Adams', 'Jefferson')
```

The first command above creates a list of presidents, while the second creates a tuple of presidents. To create the empty tuple, simply type a pair of parentheses with nothing in between.

```
>>> ()
()
```

Since parentheses are also used to clarify the order of operations in Python expressions, creating a tuple with just one element is tricky. Simply surrounding a single expression in parentheses won't work.

```
>>> (18)
18
>>> ('Madison')
'Madison'
```

Instead, include a single comma just before the closing parenthesis, which helps Python distinguish tuple parentheses from normal parentheses.

```
>>> (18,)
(18,)
>>> ('Madison',)
('Madison',)
```

You can index a tuple or apply nearly all of the list functions, operators, and methods that we have discussed in this chapter. So what, then, is the difference between a list and a tuple? Tuples can't be modified—they are immutable.

```
>>> pres_list[1] = 'Monroe'
>>> pres_list
['Washington', 'Monroe', 'Jefferson']
>>> pres_tuple[1] = 'Monroe'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Any other list functions and methods that make modifications will also fail.

```
>>> pres_tuple.remove('Washington')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'remove'
```

You can, however, use tuples in expressions that produce other values so long as the original tuple isn't modified.

```
>>> pres_tuple + ('Madison', 'Monroe')
('Washington', 'Adams', 'Jefferson', 'Madison', 'Monroe')
```

You can even modify a variable that formerly stored a tuple, again, so long as the original tuple isn't modified.

```
>>> pres_tuple = pres_tuple + ('Madison', 'Monroe')
>>> pres_tuple
('Washington', 'Adams', 'Jefferson', 'Madison', 'Monroe')
```

As you recall from the previous section, doing so modifies the variable's reference, not the tuple itself.

Note, however, that when a tuple stores a type that is copied by reference, such as a list, the contents of the list can be modified even though the reference to the list cannot. Consider the following tuple.

```
>>> tup = ([1, 2], [3, 4])
```

As we learned in the last section, the tuple contains two references—one to the list `[1, 2]` and one to the list `[3, 4]`. Since these references are part of the tuple, they cannot be modified.

```
>>> tup[0] = [5, 6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

The lists that the tuple's references point to, however, are stored in a separate location that *is* modifiable, so the individual list elements can be altered.

```
>>> tup[0][1] = 16
>>> tup
([1, 16], [3, 4])
```

Although this loophole its subtle, it is good to be aware that it exists.

Given that tuples are nearly identical to lists, why does Python include both types? Often, it is useful to ensure that data cannot be modified. Tuples provide a mechanism for passing data to other code and guaranteeing it will come back unchanged. If there is a collection of data that you know should never changed, tuples will enforce this restriction and crash your program if you accidentally violate it. In this way, tuples can serve as a guardrail, protecting you from mishandling data.

In general, you are unlikely to use tuples in your day-to-day programming. Many Python libraries produce or consume tuples, however, so it is important that you are generally familiar with their behavior.

7.10 Conclusion

In this chapter, we introduced a flexible data type that can store an unlimited amount of information, lists. We discussed creating and modifying lists, a few special list-related features, and how they are represented under the hood. Finally, we introduced tuples, which are nearly identical aside from a few added restrictions. You will find yourself using lists in nearly every program you write going forward. Storing collections of data, whether laws containing keywords relevant to a case, emails obtained during e-discovery, or SEC filings, is at the heart of the legal applications on which this book focuses.

In addition, and perhaps more importantly, this chapter introduced several concepts that will continually resurface as we introduce more advanced components of Python. Lists are the first of many data types that store other data types (e.g., a list of integers) and the first of many that can be indexed. In the process of describing lists, we delved into the distinction between functions, operators, and methods, as well as the difference between value types and reference types, concepts that apply to every Python feature we will learn going forward.

Practically and conceptually speaking, lists lay the groundwork for the array of new Python capabilities you will gain in the coming chapters. Soon, you will have a full knowledge of the core Python language and will be ready to apply your skills to a wide range of legal applications.

7.11 Cheatsheet

Lists

A list is a type of data that stores zero or more other pieces of data. The type of a list is determined by the type of data it stores. If a list stores integers, it has type "list of integers." If it stores strings, it has type "list of strings." A list can contain multiple different types of elements, but we strongly discourage this practice.

Creating Lists

A list is written by enclosing zero or more expressions in square brackets and separating the expressions with commas. For example, a list storing the integers 1, 2, and 3 is written as `[1, 2, 3]`. A list can contain a single element (e.g., `['torts']`), many elements (e.g., `[3.14, 2.79, 1.61, 6.23]`), or no elements (i.e., the empty list `[]`).

Accessing List Elements

Lists can be saved to variables.

```
>>> classes = ['torts', 'contracts', 'criminal procedure']
>>> classes
['torts', 'contracts', 'criminal procedure']
```

When it comes to lists, order matters. Each element in the list is referenced by its position (e.g.., the first item, the second item, etc.). The first item in the list is at index (that is, position number) 0. The second element in the list is at index 1, etc.. To access an element of the list, write the name of the variable that stores the list followed by a pair of square brackets containing the desired index. This is known as indexing.

```
>>> classes[0]
'torts'
>>> classes[1]
'contracts'
>>> classes[2]
'criminal procedure'
```

If you use an index that doesn't exist (e.g., index 3 or greater in the list above), the program will crash.

Negative indices count backwards from the right side of the list.

```
>>> classes[-1]
'criminal procedure'
>>> classes[-2]
'contracts'
>>> classes[-3]
```

'torts'

A negative index that goes beyond the beginning of the list will cause the program to crash.

Modifying Lists

A list element can be modified in similar fashion to a variable. Type the name of the list followed by square brackets containing the index to be modified to the left of the = operator. To the right of the = operator, type the new value to be stored at this index.

```
>>> agencies = ['ftc', 'fcc', 'dhs']
>>> agencies[1] = 'hhs'
>>> agencies
[' ftc ',  'hhs',  'dhs']
```

Nested Lists

Lists can store other lists. For example, the list

```
>>> senators = [['Cardin', 'Van Hollen'], ['Warner', 'Kaine'], ['Cornyn',
    'Cruz'], ['Feinstein', 'Harris']]
```

is a "list of lists of strings" (containing senators organized by state).

To access one of the lists contained in this list of lists, use the standard indexing procedure.

```
>>> senators[1]
[' Warner', 'Kaine']
>>> senators[3]
[' Feinstein ',  'Harris ']
```

To access a specific string stored in the list of lists of strings, index once for each level of lists.

```
>>> senators[2][0]
'Cornyn'
>>> senators[0][1]
'Van Hollen'
```

The outer first index selects the list of strings. The second index selects the particular string from within the chosen list of strings.

Slicing

Slicing is the process of creating a list that contains a shorter piece of some other list. Consider the list

```
>>> x = ['a', 'b', 'c', 'd', 'e']
```

Example slices: `['b', 'c', 'd]`, `['d', 'e']`, `['a', 'b']`, `['c']` and `[]`.

To extract a slice of a list, use the same notation as if indexing the list. Rather than using a single index, use two indices separated by a colon. The first index specifies the index at which the slice starts. The second index is one index beyond where the slice ends.

```
>>> x[1:4]
['b', 'c', 'd']
>>> x[3:5]
['d', 'e']
>>> x[0:2]
['a', 'b']
>>> x[2:3]
['c']
>>> x[1:1]
[]
```

If you leave out the index after the colon, the slice will include every element from the first index of the slice to the end of the list.

```
>>> x[2:]
['c', 'd', 'e']
```

If you leave out the index before the colon, the slice will include every element from the beginning of the list to one before the second index of the slice.

```
>>> x[:2]
['a', 'b']
```

Tuples

Tuples are lists with two key differences.

1. They are written with parentheses rather than square brackets.

   ```
   >>> x_list = [1, 2, 3, 4, 5]
   >>> x_tuple = (1, 2, 3, 4, 5)
   ```

   The empty tuple is a pair of empty parentheses: `()`. To create a tuple with one element, follow the single element with a comma inside a pair of parentheses: `(1,)` and `('hello',)`.

2. You cannot modify the elements of a tuple.

Range

The range function generates lists of integers that count. The value that range produces is something slightly different than a list, so be sure to cast the result to a list by passing it as the argument to the `list()` function.

The range function has different behavior depending on the number of arguments you provide it.

- *1 integer argument.* Produces a list containing the integers from 0 to the number one below the argument. For example, list (range(5)) produces the list [0, 1, 2, 3, 4].

- *2 integer arguments.* Produces a list containing the integers from the first argument to the number one below the second argument. For example, list (range(1, 5)) produces the list [1, 2, 3, 4].

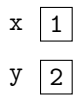
- *3 integer arguments.* Same as with two arguments, except that the third argument controls the size of the steps between list elements. For example, range(1, 10, 2) produces the list [1, 3, 5, 7, 9].

Value and Reference Types

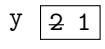
Value Types

A value type is any type of data which is copied when assigned from one variable to another. Value types: integers, floating point numbers, booleans, strings, and the none type. Example:

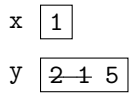
```
>>> x = 1
>>> y = 2
```




```
>>> y = x
```




```
>>> y = 5
```

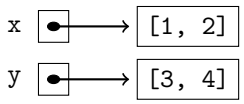


The third command copies the value of x into the separate storage space for variable y. Modifying y in the future does not change x.

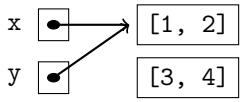
Reference Types

A reference type is any type of data which, when assigned from one variable to another, copies a reference to the same storage space. In other words, modifying the value stored by one of these variables modifies the value of the other. So far, lists are the only reference type we've encountered.

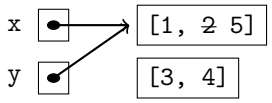
```
>>> x = [1, 2]
>>> y = [3, 4]
```




```
>>> y = x
```



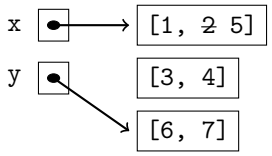

```
>>> y[1] = 5
```



Each variable stores a reference to the storage space containing its list. The third command copies the reference from x into the reference stored by y. They now refer to the same list. Modifying the list to which y refers modifies the list to which x refers.

Changing the list to which y refers does not affect x.

```
>>> y = [6, 7]
```



Operators, Functions, and Methods

Operator. A symbol like + and ∗ that operate on expressions to their left and right to create new values.

Function. Takes one or more expressions as arguments and operates on them to produce a new value. By convention, functions typically do not modify their arguments (although this is not a strict rule and there are exceptions).

To call (that is, execute) a function, write the name of the function followed by a set of parentheses. Within the parentheses, enclose the function's arguments separted by commas (order matters). The function call will evaluate to the value that the function computes.

For example, to compute the length of a list with the len function and the argument `['a', 'b', 'c', 'd']`:

```
>>> len(['a', 'b', 'c'])
3
```

Method. Operates on an object and takes one or more expressions as arguments. By convention, method calls often modify their object and sometimes evaluate to a value. Other than the presence of the object and a slightly different convention, methods are identical to functions.

To call a method, write the name of the object, a dot, and the name of the method followed by a set of parentheses. Within the parentheses, enclose the method's arguments separated by commas (order matters). The method call will modify the object (if the method does so) and evaluate to a value (if the method does so).

For example, to remove the last element from a list with the pop method:

```
>>> x = ['a', 'b', 'c']
>>> x.pop()
'c'
>>> x
['a', 'b']
```

The pop method call's object was the variable x (which stores a list) and the method was called with no arguments. The method call evaluated to the last element in the list (`'c'`). It modified x, removing the last element.

List Operators

Op.	Operands	Description	Example
`+`	Two lists	Concatenates the lists, joining them together into a new list.	`[1, 2] + [3, 4]` evalutes to `[1, 2, 3, 4]`
`*`	A list An integer	Replicates the list, concatenating it to itself the number of times specified by the integer.	`[1, 2] * 3` evalutes to `[1, 2, 1, 2, 1, 2]`
`in`	An element A list	Evaluates to `True` if the element is in the list and `False` otherwise	`1 in [1, 2, 3]` evaluates to `True` `4 in [1, 2, 3]` evaluates to `False`
`not in`	An element A list	Evaluates to `False` if the element is in the list and `True` otherwise	`1 not in [1, 2, 3]` evaluates to `False` `4 not in [1, 2, 3]` evaluates to `True`
`del`	An indexed list	Modifies the list, deleting the value at the specified index.	if `x` stores `['a', 'b', 'c']`, the statement `del x[1]` changes `x` into `['a', 'c']`.

List Functions

Name	Arguments	Description	Example
`len`	1. A list.	Evaluates to the integer length of the list.	`len([4, 5, 6])` evaluates to `3`.
`min`	1. A list of comparable elements	Evaluates to the smallest item in the list.	`min([3, 1, 4])` evaluates to `1`.
`max`	1. A list of comparable elements	Evaluates to the largest item in the list.	`max([3, 1, 4])` evaluates to `4`.
`sorted`	1. A list of comparable elements	Evaluates to a list containing the same elements sorted in ascending order.	`sorted([3, 1, 4])` evaluates to `[1, 3, 4]`.

List Methods

Name	Object and Arguments	Description	Example
`extend`	Object: A list. 1. A list.	Concatenates the argument onto the end of the object.	If `x` stores `[1, 2]`, `x.extend([3, 4])` changes `x` to `[1, 2, 3, 4]`
`append`	Object: A list. 1. An element.	Adds the element as the last item of the object list.	If `x` stores `[1, 2]`, `x.append(3)` changes `x` to `[1, 2, 3]`
`pop`	Object: A list.	Removes the last element of the object. Evaluates to the removed element.	If `x` stores `[1, 2, 3]`, `x.pop()` changes `x` to `[1, 2]` and evaluates to `3`
`pop`	Object: A list. 1. An index.	Removes the element of the object at the specified index. Evaluates to the removed element.	If `x` stores `[1, 2, 3]`, `x.pop(1)` changes `x` to `[1, 3]` and evaluates to `2`
insert	Object: A list. 1. An index. 2. An element.	Inserts the element into the object at the specified index, shifting all elements afterwards down by one index.	If `x` stores `['a', 'b', 'c']`, `x.insert(2, 'j')` changes `x` to `['a', 'b', 'j', 'c']`
remove	Object: A list. 1. An element	Removes the first appearance of the element from the object.	If `x` stores `[3, 1, 4, 1, 5]`, `x.remove(1)` changes `x` to `[3, 4, 1, 5]`
index	Object: A list. 1. An element.	Evaluates to the index at which the element first appears in the object. If the element does not appear, Python crashes.	If `x` stores `['a', 'b', 'c']`, `x.index('b')` evaluates to `1`.
count	Object: A list. 1. An element.	Evaluates to the number of times the element appears in the object.	If `x` stores `[3, 1, 4, 1, 5]`, `x.count(1)` evaluates to `2`.
`sort`	Object: A list of comparable elements.	Modifies the object, placing its elements in ascending order	If `x` stores `[3, 1, 4]`, `x.sort()` changes `x` to `[1, 3, 4]`
`reverse`	Object: A list of comparable elements	Reverses the order of the elements stored in the object.	If `x` stores `[1, 2, 3]`, `x.reverse()` changes `x` to `[3, 2, 1]`

Keywords

Element—A single item stored in a list.

Empty List—A list with no entries.

Immutable—Data that cannot be modified. Immutability is often a useful guardrail to ensure that you don't accidentally modify something you shouldn't. Example: A string is immutable; Python will crash if you try to modify one of its characters as if it were a list.

Index—Noun: a particular location within a list. Example: In the list `['a', 'b', 'c']`, `'a'` is at index 0 and c is at index 2. Verb: to extract an element from a list at a particular location (see: indexing).

Indexing—The process of extracting an element from a list using its index. Example: the expression `x[2]` represents indexing the list x to retrieve the element at index 2.

Slice—Noun: a small piece of a larger list. Example: `[3, 4]` is a slice of `[1, 2, 3, 4]`. Verb: to extract a slice from a list.

Subscript—See: index.

Subscripting—See: indexing.

Zero-indexed—The starting index is 0. Python lists are zero-indexed.

Chapter 8

Strings

For the purposes of this book, strings are perhaps the most important type of data. Strings represent words, sentences, and text. They can capture Supreme Court opinions, regulatory filings, web pages, and spreadsheets. Many of the most tantalizing legal applications we will discuss in later chapters revolve around strings in one form or another.

Although we introduced the basics of strings in Chapter 4, we will spend this chapter exploring many advanced capabilities that make sense only in the context of lists. As you will soon see, Python treats strings as lists of characters, so nearly everything we have learned about lists also applies here. This fact should explain why many of the string operations from Chapter 4, such as the len function and the + operator, also worked on lists in the previous chapter.

With that said, there are two key differences between lists and strings. First, like tuples, strings are immutable—you can't modify the contents of a string once you create it. Second, Python includes a range of string-specific methods above and beyond those available for lists, operations designed specifically with textual data in mind. You can check whether a string consists entirely of alphanumeric characters, test whether a string starts or ends with a particular sequence, and look for one string inside of another.

Strings are at the heart of most of the legal applications we discuss throughout this book. This chapter focuses on the advanced mechanics of strings, but we will return to them again later when we study regular expressions, which provide a powerful way to search for patterns in large bodies of textual data.

8.1 Strings are Lists

Nearly everything we studied in the previous chapter applies to strings. Python treats strings as tuples—they can be indexed, analyzed, and manipulated just like lists, but they cannot be modified. Like lists, strings are copied by reference, but, since they cannot be modified, this distinction is of no practical

consequence.[1] We will spend this section exploring list operations in the context of strings.

8.1.1 Indexing and Slicing

You can access individual characters of a string using the same square bracket notation as for indexing lists. When you extract a character, it is stored as a string of length 1 containing just that character. Recall that Python starts counting at zero, so the first character of a string is at index zero, and the last character is at an index one less than the length of the string.

```
>>> hello = 'Hello, world!'
>>> hello[0]
'H'
>>> hello[1]
'e'
>>> hello[8]
'w'
>>> hello[len(hello) - 1]
'!'
```

As with lists, you will see an error message if you supply an index that goes beyond the end of the string.

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

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

Negative indices work exactly the same way as with lists. They start counting from the right side of the string, where the rightmost character is at index -1.

```
>>> hello[-1]
'!'
>>> hello[-6]
'w'
>>> hello[-len(hello)]
'H'
```

[1] Strings are indeed reference types. When a string is copied from one variable to another, the two variables contain references to the same storage area in the memory of your computer. The typical pitfall when dealing with values that are copied by reference is that, when two variables contain references to the same value, modifying one variable modifies the other. Since strings cannot be modified, you don't have to worry about this problem when dealing with strings. Hence, the distinction between a value type and a reference type doesn't particularly matter when it comes to strings (and any other immutable value).

As we mentioned before, strings are immutable, so Python will crash your program if you try to modify a character.

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

You can slice strings to extract substrings—smaller pieces of a larger string. Recall that the first index of a slice specifies where it should start and the second index specifies one character beyond where it should end.

```
>>> hello[0:5]
'Hello'
>>> hello[7:12]
'world'
```

Recall that, if you keep the colon but leave off the first index, the slice will start at the beginning of the string; if you leave off the last index, it will finish at the end of the string.

8.1.2 Operators

All of the list operators are also valid on strings. You can concatenate two strings, joining them together into a single string, with the + operator. You can replicate a string, repeating it several times over, with the ∗ operator.

```
>>> 'Hello, ' + 'world!'
'Hello, world!'
>>> 'Hello' * 4
'HelloHelloHelloHello'
```

The concatenation (+) operator is by far the most useful string operator, and you will use it very frequently in practice.

The in and not in boolean operators are similarly important. They allow you to test whether a character, or more importantly, an entire substring, is contained within a string. For example, you can ask whether the string 'Hello' appears within the larger string 'Hello, world!'

```
>>> 'Hello' in 'Hello, world!'
True
>>> 'world' in 'Hello, world!'
True
>>> 'Cello' in 'Hello, world!'
False
>>> 'Cello' not in 'Hello, world!'
True
```

The in and not in operators are case sensitive, so they will find a match only if the cases line up perfectly. For example, the string 'hello' will not be found within the string 'Hello, world!'

```
>>> 'hello' in 'Hello, world!'
False
```

8.1.3 Functions

Of all the list functions we studied in the previous chapter, only len is particularly useful. The len function evaluates to the length of its string argument.

```
>>> len('Hello')
5
>>> len('Dilapidated')
11
>>> len('')
0
```

Other list-related functions successfully run without leading to errors, but there are a few settings in which it is helpful to find the minimum or maximum character of a string or to rearrange the characters in sorted order.

8.1.4 Methods

Since many of the list methods make modifications to their objects, they lead to errors when used on strings, which are immutable. Two methods, however, are quite convenient. The list count method tallies the number of occurrences of a particular item in a list. In the context of strings, it can be used to find the number of times a character or substring appears in a string.

```
>>> hamlet = 'To be or not to be'
>>> hamlet.count('o')
4
>>> hamlet.count('be')
2
```

Since strings are case sensitive, the count will include only matches that were identical to the search string.

```
>>> hamlet.count('to')
1
>>> hamlet.count('To')
1
```

The index method will return the starting index of the first appearance of a character or substring.

```
>>> hamlet.index('o')
1
>>> hamlet.index('be')
3
>>> hamlet.index('to')
13
```

Recall that, if no matches are found, the index method will produce an error.

```
>>> hamlet.index('Yorick')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

8.2 Creating Strings

In Chapter 4, we described the most basic way of creating a string—enclosing it in single quotes. Python provides a few tricks for avoiding some of the limitations of this approach, including several other ways to create strings.

One problem with using single quotes to mark the beginning and end of a string is that you can't use any single quotes inside of the string. If you do, it will be impossible for Python to figure out which quotes demarcate the boundaries of the string and which quotes are part of the string itself.

```
>>> 'Don't worry, be happy'
  File "<stdin>", line 1
    'Don't worry, be happy'
         ^
SyntaxError: invalid syntax
```

Python assumes that the apostrophe in the word don't is the closing quote of the string and is unable to make sense of the characters that follow. One way of getting around the problem is to enclose the string in double quotes instead.

```
>>> "Don't worry, be happy"
"Don't worry, be happy"
```

Double quotes are an equally valid way of creating a string. Stylistically speaking, either kind of quote is acceptable so long as you are consistent.

Unfortunately, even switching to a different kind of quote is insufficient if you need to use both single and double quotes within your string.

```
>>> '"Don't worry, be happy" by Bobby McFerrin'
  File "<stdin>", line 1
    '"Don't worry, be happy" by Bobby McFerrin'
          ^
SyntaxError: invalid syntax
```

To get around this problem, Python makes it possible to selectively deprive an appearnce of a character that has special meaning in Python (e.g., a quote character) of this special meaning by preceding it with a backslash (\). When you do so, Python will treat the character as if it were any other letter of the alphabet, ignoring its behavior (e.g., as a marker of a string's beginning or end). The process of using backslashes to this effect is known as escaping. To escape a character, simply precede it with a backslash (\). An escaped single quote would be written as \'.

We can rewrite the string that Python rejected before by escaping the problematic apostrophe.

```
>>> 'Don\'t worry, be happy'
"Don't worry, be happy"
```

Now that the apostrophe has been escaped, Python knows not to interpret it as the end of the string and continues reading until the unescaped single quote that we intend to serve this purpose. Notice that, when Python echoes the string back, it does so with double quotes so as to avoid the need for escaping it. When a string contains both single and double quotes, however, Python does not have this flexibility and echoes back the escape characters.

```
>>> '"Don\'t worry, be happy" by Bobby McFerrin'
'"Don\'t worry, be happy" by Bobby McFerrin'
```

Backslashes are not part of the string itself. They do not count toward its length and they cannot be indexed. In the string above, for example, the fourth character (the character at index 3) is the apostrophe. The string `'\''`—the string that contains just an escaped single quote—has length 1.

```
>>> len('\'')
1
```

As far as Python is concerned, the escape character is just a visual convenience that makes it easier for you to communicate your intent. When Python echoes strings during interactions at the Python shell, it reproduces the escape character only for the sake of human readability.

When this string is actually displayed with a print statement, the escape character will disappear.

```
>>> s = 'shan\'t'
>>> s[4]
"'"
>>> print(s)
shan't
```

Python includes a variety of other escape sequences. You can escape double quotes in the same manner as single quotes.

```
>>> "\"Don't worry, be happy\" by Bobby McFerrin"
"\"Don't worry, be happy\" by Bobby McFerrin"
```

Just as before, Python treats the escaped double quotes as normal characters with no special meaning.

But what if you want to create a string that includes the backslash character? You can escape that too with the sequence `\\`.

```
>>> 'The Python escape character is \\'
'The Python escape character is \\'
>>> print('The Python escape character is \\')
The Python escape character is \
```

As before, the escaping disappears when the string is printed.

Python uses escaping to fulfill a second purpose—it allows you to write characters that would otherwise be impossible to express within a normal string or even on a normal keyboard. For example, you can use the escape sequence \n to create a line break within a string.

```
>>> 'Line 1\nLine 2'
'Line 1\nLine2'
>>> print('Line 1\nLine 2')
Line 1
Line 2
>>> print('An extra blank line at the end of this print\n')
An extra blank line at the end of this print

>>> print('Lots of space\n\n\nBetween these lines')
Lots of space


Between these lines
```

Python processes each \n escape sequence into a blank line when a string is printed.

You can create tab characters in a similar manner with the escape sequence \t. The example below uses both the \t and \n escape sequences.

```
>>> '\tOne tab\n\t\tTwo tabs\n\t\t\tThree tabs'
        One tab
                Two tabs
                        Three tabs
```

There are numerous other escape sequences for creating all sorts of special characters. For example, the escape sequence \u00E9 will create a é character. Every single character (including emojis, assuming Python is updated to support them) has an escape sequence. These sequences are universally agreed upon in a standard called unicode, which specifies a definitive list of characters from the alphabets of various languages, other symbols, and emojis that must be digitally printable. You can look up the unicode numbers and Python escape sequences for these characters online. We have found www.fileformat.info/info /unicode/char/ to be a useful resource.

Occasionally, you will need to write a Python string that includes numerous backslashes that are not intended to function as escape characters. For example, you might want to write the sentence,

To escape an escape character, type \\.

To print this sentence in Python, you would need to write the string

```
'To escape an escape character, type \\\\.'
```

since each backslash needs to be escaped by another backslash. The number of backslashes can add up quickly and become confusing for a reader. This escaping is an equally large problem when trying to write file paths in Windows, which separates each directory with a backslash. For example, to write the file path C:\users\Larry\documents, you would need to write the string

```
'C:\\users\\Larry\\documents'
```

escaping each backslash character that appears.

Although these scenarios might seem contrived at first glance, it arises quite often in practice. Regular expressions, which make it possible to glean information from large databases of text, rely on these non-escape backslashes. We will study them in detail later in this book.

When you expect to use a large number of non-escape backslashes, you can create a special string that disables all escaping by preceding the opening ' or " with the letter r. The above string could be rewritten

```
r'To escape an escape character, type \\'
```

This kind of string is known as a raw string, since its contents are interpreted exactly as they appear—without any escaping. You can use raw strings to similar effect with other escape sequences, replacing

```
'\\n and \\t are also escape sequences'
```

with

```
r'\n and \t are also escape sequences'
```

The first character of the string is the backslash (\). The second character is the letter n. When this string is printed, the characters will appear exactly as they are in the string.

```
>>> print(r'\n and \t are also escape sequences')
\n and \t are also escape sequences
```

Beyond escaping, Python provides another mechanism for writing strings that include new lines, tabs, and other whitespace. You can create a string that stretches across multiple lines by triple-quoting—beginning and ending the string with three quotes (single or double, as long as all three quotes are the same).

```
>>> '''First line
... second line
...
... fourth line (third is blank)
... end'''
'First line\nsecond line\nfourth line (third is blank)\nend'
```

The Python shell will automatically create the ellipses above if you press enter or return before you have closed a triple-quoted string. For comparison, below is a triple-quoted string in an actual Python program.

```
# The string representing the bill for an estate plan.
bill = '''
Will ..... $650
Trust ..... $200
Power of Attorney...$300
------------------------
Total $1,150'''

print(bill)
```

Creating the same string with escape characters for newlines would become a confusing jumble of characters. Triple-quoting makes the string far easier to create, edit, and understand. The string is written in Python with the spacing it should contain when printed. You can triple quote with either single (''') or double (""") quote characters so long as you open and close with string with the same pattern.

8.3 String-Specific Methods

Although strings can be reduced to lists of characters, the words and sentences that they contain are capable of conveying meaning well beyond that found in a list of integers. In other words, strings are more interesting than your average list because they contain text. In acknowledgement of this fact, Python provides a vast number of string methods that go well beyond those available for ordinary lists. They examine or manipulate qualities that make sense only in the context of text.

8.3.1 Tests

Python provides several methods for classifying the contents of a string. Does it contain only numbers? Only letters? Only lowercase letters? These methods are particularly useful for input validation—checking to make sure a user provided the kind of data you expected. For example, imagine that a tax calculation program asks a user for their annual income and receives the response 'trampoline'. If the program tried to convert this string into a number, Python would crash. It would be far better to check that the user's input was really a valid number before trying to do math with it. To do so, you can use the isdigit method, which checks whether a string consists entirely of characters within the digits 0 through 9. It evalutes to boolean True if so and False otherwise.

```
>>> '123'.isdigit()
True
>>> 'trampoline'.isdigit()
False
>>> '3.14'.isdigit()
False
```

```
>>> '867-5309'.isdigit()
False
>>> '$895'.isdigit()
False
```

Suppose you are writing a program that calculates a user's income tax. The input might be collected in a manner similar to that below.

```
1 # Collect the user's annual income.
2 income = input('What is your annual income? $')
3
4 # Keep looping until the user provides valid input.
5 while not(income.isdigit()):
6     income = input('Invalid number. Try again. $')
7
8 # Convert the number to an integer.
9 income_int = int(income)
```

This program asks for the user's income on line 2. The loop on lines 5 and 6 will execute only if the user neglected to supply a number in response to the request for input. In this case, the statement on line 6 will continue to ask the user for input until it is a valid number. The code on line 9 can therefore be certain that the value of income is convertible into an integer since it must consist only of digits.

Similar functions exist for several other kinds of data. The isalpha method tests whether a string consists only of letters. The isalnum checks whether a string is alphanumeric—whether it consists only of a combination of letters and numbers.

```
>>> 'hello'.isalpha()
True
>>> 'thing1'.isalpha()
False
>>> 'thing1'.isalnum()
True
```

You can test whether a string consists only of uppercase or lowercase letters with the isupper and islower methods.

```
>>> 'hello'.islower()
True
>>> 'Hello'.islower()
False
>>> 'Hello'.isupper()
False
>>> 'HELLO'.isupper()
True
```

Python provides several other similar tests (isspace, istitle , etc.) although we have discussed those that are most useful.

8.3.2 Capitalization

You can alter the capitalization of a string using the upper and lower methods. Both are self-explanatory—the former produces a version of the string that is entirely in uppercase and the latter a version that is entirely in lowercase. Neither of these methods modifies the original string (since doing so is impossible); instead, they evaluate to new strings that are identical to the old ones except with the updated capitalization.

```
>>> 'hello'.upper()
HELLO
>>> 'Supreme Court'.upper()
SUPREME COURT
>>> 'Supreme Court'.lower()
supreme court
```

These methods are convenient for comparing strings in a case-insensitive way. If you recall, the == and != operators for strings are case-sensitive. According to these operators, the string 'hello' is different than the string 'Hello' because they have different capitalization. In many situations, you will want to compare strings in a case-insensitive manner—you will want your program to treat 'hello' and 'Hello' as the same string. To achieve this behavior, convert both strings to uppercase or lowercase before comparing them. In doing so, you will erase any capitalization differences between the strings.

```
>>> s1 = 'hello'
>>> s2 = 'Hello'
>>> s1 == s2
False
>>> s1.lower() == s2.lower()
True
>>> s1.upper() == s2.upper()
True
```

Python also includes a method called `capitalize`, which capitalizes the first character of a string.

8.3.3 Formatting

Strings are often used to present information, and Python includes a number of methods that help you format strings in more visually appealing ways.

The most important string formatting method is conveniently known as format. This method allows you to leave empty fields in a string and ask Python to fill them in like a form (or a Mad-Lib). For example, you could write

```
'Hello, {0}!'.format('Bob')
```

to produce the string

```
'Hello, Bob!'
```

Here's how the format method works. First, you create a string (e.g., `'Hello, {0}'`). Wherever you want to leave fields be filled in with data later, you write two curly braces with a number in between (e.g., `'{0}'`). When you call the format method, Python will fill the method's arguments (e.g., `'Bob'`) into the places where you put the curly braces before (e.g., `'Hello, Bob'`). The number inside the curly braces specifies which argument you want to appear there—the first argument is argument 0, the second argument is argument 1, etc. The arguments don't have to be strings—Python automatically calls the str function to perform the necessary conversions.

Consider the following example: you want to create a form letter for wishing people a happy birthday. First, you create a string that includes two fields to filled in later—the person's name and their age.

```
>>> birthday_form = 'Happy {1}{2} birthday, {0}!'
```

For each person whom you want to wish a happy birthday, call the format method with the appropriate arguments for each field in the proper order.

```
>>> birthday_form.format('Larry', 45, 'th')
Happy 45th birthday, Larry!
>>> birthday_form.format('Chelsea', 21, 'st')
Happy 21st birthday, Chelsea!
>>> birthday_form.format('Millard', 62, 'nd')
Happy 62nd birthday, Millard!
```

Notice that the format function does not modify the original string; instead, it evaluates to a new string with the fields appropriately substituted into the form.

Why go through all of the trouble to do this formatting? The alternative approach is far messier. You would have to use the + operator multiple times to concatenate all of the values together and use the str function to convert any numerical values to strings. The above commands would have to be rewritten as:

```
>>> 'Happy ' + str(45) + 'th' + ' birthday, ' + 'Larry' + '!')
Happy 45th birthday, Larry!
```

Not only is this command messy, but it isn't reusable. The variable `birthday_form` makes it easy to create a form once and reuse it for each new combination of person and age. In contrast, the version that uses manual concatenation would have to be copied and pasted over and over again.

The format method is a huge convenience that will save you time and protect you from mistakes. Here, we have only scratched the surface of its capabilities. You can specify more complicated substitution patterns that round numbers to a certain number of decimal places or right-justify substituted text. If you are interested in learning more, you can consult the official Python documentation for an exhaustive list of every feature.

Python includes a number of other functions that in some way format strings. For example, you can justify a string to the left, right, or center using the ljust , rjust , and center methods. These methods take as a single argument the

width of the space within which you want to justify the string (as measured in characters). For example, writing

```
s.center(30)
```

would center the string stored in the variable s on a line 30 characters long. It does so by filling in spaces to the left and right of the string as necessary to ensure it is centered.

```
>>> s = 'hello'
>>> s.center(17)
'      hello      '
```

As before, the string itself is not modified (since strings are immutable); instead, a new string is created. Since it's a little difficult to see the spaces that Python uses to justify a string, we can take advantage of an additional capability. You can provide a second argument that specifies the character that should be used to create the justification. For example, you could provide an asterisk character.

```
>>> s.center(17, '*')
'******hello******'
```

Changing this second argument makes the precise effects of the center method much easier to see. Below, we can see the effects of using the ljust and rjust methods.

```
>>> s.ljust(17)
'hello            '
>>> s.ljust(17, '*')
'hello**********'
>>> s.rjust(17)
'            hello'
>>> s.rjust(17, '*')
'**********hello'
```

Python includes three similar functions that have the opposite effect, stripping spaces or characters off of the edges of strings to, for example, turn the string ' hello ' into 'hello'. This task often serves as a key step in the process of data sanitization that ensures you are working with "clean" data. We previously discussed how you will sometimes be interested in comparing strings in a case-insensitive manner (where 'hello' and 'Hello' are the same); similarly, there are many settings where you will want to ignore extra spaces on either side of a string. To do so you can use the strip method, which removes all whitespace from the left and right edges of a string.

```
>>> ' hello '.strip()
'hello'
>>> '\n\t \n hello\t\t\t \n'.strip()
'hello'
```

Python also provides the methods lstrip and rstrip when you want to remove whitespace from only the left or right sides of a string respectively. You can provide any of these methods with an optional argument that specifies which characters you want to strip.

```
>>> ' hello '.strip('*')
'  hello  '
>>> '***hello*****'.lstrip('*')
'hello*****'
```

The argument can consist of more than one character. If it does, Python will strip *any* of the characters from that string. The order of characters in the argument doesn't matter.

```
>>> ':-_hello_1-_-:'.strip('-_:')
'hello_1'
```

The strip methods work their way from the outside inwards, stopping as soon as they see a character that they are not allowed to remove. In the call to strip above, the method is permitted to remove only the '-', '_', and ':' characters, so it stops at the '1' character to to the right of 'hello', meaning it never gets to the '_' between hello and 1.

8.3.4 Joining and Splitting

Python includes two methods, join and split, that make it easy to concatenate a list of strings into a single string or the reverse. The join method takes as its sole argument a list of strings. They are concatenated together, with the string object on which the join method was called inserted between each item in the list. For example, the expression

```
'-'.join(['warren', 'black', 'frankfurter'])
```

would evaluate to

```
'warren-black-frankfurter'
```

We could have chosen to combine the list of strings with any string we liked, including words or new lines.

```
>>> ' and '.join(['warren', 'black', 'frankfurter'])
'warren and black and frankfurter'
>>> '\n'.join(['warren', 'black', 'frankfurter'])
'warren\nblack\nfrankfurter'
>>> print('\n'.join(['warren', 'black', 'frankfurter']))
warren
black
frankfurter
```

If we want the list of strings to be combined without anything inserted in between, we can use the empty string.

```
>>> ''.join(['warren', 'black', 'frankfurter'])
'warrenblackfrankfurter'
```

The split function does precisely the opposite—it divides its string object into a list of strings, breaking it up whenever it encounters whitespace (including tabs and new lines).

```
>>> 'warren black frankfurter\n stewart'.split()
['warren', 'black', 'frankfurter', 'stewart']
```

If you want to split along a particular pattern in the string, you can do so by providing it as an argument to the split method.

```
>>> 'warren, black, frankfurter, stewart'.split()
['warren,', 'black,', 'frankfurter,', 'stewart']
>>> 'warren, black, frankfurter, stewart'.split(', ')
['warren', 'black', 'frankfurter', 'stewart']
```

When the justices' names are separated by commas, calling split without any arguments does not have the intended effect. The first three justices will have commas at the end of their names because the split function separated only where it found whitespace. To solve this problem, we provided the split function with an argument—the string ', ', ensuring that the resulting list included only the justices' names.

You will frequently find splitting along the newline character ('\n') a useful way to break a multi-line string into a list of lines.

```
>>> justices = '''earl warren
... hugo black
... felix frankfurter'''
>>> justices.split('\n')
['earl warren', 'hugo black', 'felix frankfurter']
```

Had you tried to call split without any arguments, it would have split at each line break, but it would have also split at the spaces between the justices' first and last names.

```
>>> justices.split()
['earl', 'warren', 'hugo', 'black', 'felix', 'frankfurter']
```

Note that, when you specify the argument for split, it will split every time it sees the argument, even if there is nothing in between. Consider the following command:

```
'warren*black**frankfurter'.split('*')
```

We have asked Python to split the string at every asterisk. Between the last names of Justices Black and Frankfurter, however, there are two asterisks. Here is the result Python produces:

```
['warren', 'black', '', 'frankfurter']
```

Python divided the string at every asterisk including the spot where there were two in a row. Since there were no other characters between the consecutive asterisks, Python generated the empty string (the third item in the list above).

The split method, although exceedingly useful, is still quite rigid. You must specify an exact pattern, and Python will split along that pattern alone. When we study regular expressions, we will learn how to split flexibly along patterns instead.

8.3.5 Searching

When it comes to strings, one of the most common questions to ask is whether one string contains another. This question spans making sense of thousands of comments on FCC rulemakings, summarizing Supreme Court hearings, or performing e-discovery. The Google search engine is the product of applying this idea to the entire Internet. We will spent an entire chapter on searching strings with regular expressions, but we will summarize Python's built-in methods for doing so here.

We have already seen the in and not in operators for lists and strings, which check whether the string to the left appears within the string to the right and evaluates to a boolean value. Python provides two similar methods, startswith and endswith, which check whether a string begins or ends with another string. Both methods behave exactly as expected—they evaluate to True if the string starts/ends with the method's argument or False otherwise.

```
>>> hello = 'Hello, world!'
>>> hello.startswith('Hello')
True
>>> hello.startswith('Goodbye')
False
>>> hello.endswith('!')
True
>>> hello.endswith('world!')
True
>>> hello.startswith('world')
False
```

As with all string methods, startswith and endswith are case-sensitive.

```
>>> hello.startswith('hello')
False
>>> hello.startswith('World!')
False
```

As before, you can overcome this case-sensitivity by converting both object and argument to the same case.

```
>>> hello.upper().endswith('World!'.upper())
True
```

Python also provides a string-specific way of finding the index at which one string appears inside of another via the find method. The find method takes one argument, the string you want to search for (sometimes referred to as the needle). It returns the index at which the needle appears within the method's object (correspondingly referred to as the haystack).

```
>>> s = 'mellow yellow'
>>> s.find('low')
3
```

So far, this method is identical to the list index method we have already discussed. The difference is that, if the needle isn't found, the find method returns the integer −1 rather than crashing as index does.

```
>>> s = 'mellow yellow'
>>> s.find('high')
-1
```

The final searching method we will discuss in this section is replace, which takes two strings as its arguments. It creates and evaluates to a new string where all instances of the first argument have been replaced with the second. For example, the expression

```
s.replace('hello', 'goodbye')
```

would evaluate to a copy of the string stored in s where every instance of 'hello' was replaced with 'goodbye'.

```
>>> before = 'Senator Kennedy, Senator Nixon, Senator Obama'
>>> after = before.replace('Senator', 'President')
>>> after
'President Kennedy, President Nixon, President Obama'
```

Once again, the replace method does not change the string itself because strings are immutable. As we saw in the last chapter, methods of lists tend to change the lists. Methods of strings cannot, so they typically return a new string, which can be assigned to a new variable.

When you want to delete all instances of one string from another, you can use the empty string ('') as the second argument for replace. This tells Python to replace all instances of the first argument with the empty string—that is to say, it tells Python to delete all instances of the first argument.

```
>>> after = before.replace('Senator ', '')
>>> after
'Kennedy, Nixon, Obama'
>>> after.replace(',', '')
'Kennedy Nixon Obama'
```

8.4 Lists and Text Files

In Section 4.4, we introduced the concept of text files. By storing strings in text files, you can give your programs the ability to retain information even after they are finished executing. Files have another important purpose that becomes relevant now that our Python programs can use lists to manage arbitrarily large amounts of data. Our programs will often operate on collections of data—like the list of all Supreme Court opinions that have ever been written—that are far too large to write directly into Python files or manually enter as input. These datasets will often be stored in massive text files that serve as the source of information for the analyses that our programs run. To be able to operate on these datasets, we will need to know how to store and retrieve lists from text files.

Recall that, to manipulate a text file, you first need to open it with the open function. The open function takes two arguments: the string name of the file you wish to open and a string that specifies whether to open the file in read-mode (`'r'`) or write-mode (`'w'`).

```
>>> fh = open('presidents.txt', 'w')
```

The open function returns a filehandle object that can be used to manipulate the file that was open. Above, we save this filehandle to the variable fh. Recall that opening a file in write-mode creates the file if it doesn't exist and wipes out the existing file if it does.

Now that we have a filehandle open in write-mode, we can use various methods to write to the file. In Chapter 4, we used the write method to write individual strings to text files. Here, we will focus on storing lists of strings in files.

A text file stores one long string, so how should we store a list of strings? One standard approach is to store each item of the list on its own line. Concretely, if we had a list of strings containing the names of each president, the text file presidents.txt should look as below:

```
Washington
Adams
Jefferson
...
```

We can achieve this effect using the join method. First, we need a list of strings.

```
>>> presidents = ['washington', 'adams', 'jefferson']
```

Then, we need to combine this list of presidents into one long string where each president is on its own line. In other words, we want to concatenate the names of the presidents together with a newline character between each pair of names. The join method allows us to do just that.

```
>>> president_str = '\n'.join(presidents)
>>> president_str
'washington\nadams\njefferson'
```

We call the join method on the object '\n'—a string containing the newline character. Its argument is the list of strings we want to store to the file. The result of this join operation is a single string containing the strings from our list concatenated together and separted by copies of the newline character. We can then write this string to the file.

```
>>> fp.write(president_str)
>>> fp.close()
```

We can verify that the list was written properly by reading it back from the file.

```
>>> president_str2 = open('presidents.txt').read()
>>> president_str2
'washington\nadams\njefferson'
```

To obtain the original list, we can use the split method to break the string at each newline character.

```
>>> president_str2.split('\n')
['washington', 'adams', 'jefferson']
```

For convenience, we can combine these steps together. When writing, we need to open the file, write the list, and then close.

```
>>> fp = open('presidents.txt', 'w')
>>> fp.write('\n'.join(presidents))
>>> fp.close()
```

We can pass the result of join directly to the write method.

We can read the list back in a single line.

```
>>> open('presidents.txt').read().split('\n')
['washington', 'adams', 'jefferson']
```

This statement chains together a number of function and method calls. First, the call to open opens the file in read-mode. The open function returns a filepointer that becomes the object for the call to read, which returns the string contents of the file. The call to split uses this string as its object and returns a list of strings separated along the newline character.

Notice that this method of saving a list to a file doesn't work if the strings themselves contain newline characters. Still, in many situations, this method of storing a list of strings to a file is an effective strategy. When a string does contain newline characters, you may be able to use a different character as a separator, such as a comma.

8.5 Conclusion

In this chapter, we have explored strings in extreme detail. Although much of this content was review from earlier discussions of strings or lists, you will likely find this comprehensive catalog of string operations useful down the road. As

a lawyer, the most important data you will work with is text. Even content that might not seem like text initially, such as web pages or spreadsheets, are actually strings underneath. In short, a thorough working knowledge of strings will be a valuable asset, so we believe that this exhaustive review will come in handy later.

We will revisit strings again when we introduce regular expressions. Throughout this chapter, we have hinted that regular expressions offer a more powerful way to achieve many of the same goals. Rather than testing whether one string appears within another, regular expressions offer a way to look for flexible patterns that better mimic the irregularity of natural language.

In the meantime, however, we will continue our systematic overview of methods for organizing data and programs in Python. In the following chapter, we will discuss for-loops, which are a special kind of loop purpose-built for collections of data like lists and strings.

8.6 Cheatsheet

Strings are Lists

Python treats each string as a list of its constituent characters (represented as one-character strings). Unlike lists, strings are immutable—their elements cannot be modified.

The individual elements of strings can be accessed using list indexing. Substrings (slices of strings) can be accessed using list slicing.

Creating Strings

- Strings can be enclosed in either single or double quotes.
- To include a single or double quote inside a string, escape it by preceding it with a backslash (\). For example, the string `'don\'t'` is a string with five characters whose fourth element is the `'` (single quote) character.
- There are special escape sequences to represent characters that otherwise can't be typed. The sequence \n is a single character representing a new line and the sequence \t is a single character representing a tab.
- To type a string over multiple lines, open and close the string with a triple quote (either `'''` or `"""`).
- To type a string that ignores escaping rules and treats \ characters as \ characters, precede the opening quote of the string with the letter r.

Manipulating Strings

Operators

Op.	Operands	Description	Example
`+`	Two strings	Concatenates the strings.	`'Abe ' + 'Lincoln'` evalutes to `'Abe Lincoln'`
`*`	A string An integer	Replicates the string, concatenating it to itself the number of times specified by the integer.	`'hi' * 3` evalutes to `'hi hi hi'`
`in`	A string needle A string haystack	Evaluates to `True` if the needle is a substring of the haystack and `False` otherwise. This operator is case-sensitive	`'an' in 'banana'` evaluates to `True` `'anna' in 'banana'` evaluates to `False`
`not in`	A string needle A string haystack	Evaluates to `False` if the needle is a substring of the haystack and `False` otherwise. This operator is case-sensitive	`'an' not in 'banana'` evaluates to `False` `'anna' not in 'banana'` evaluates to `True`

Functions

Note: All list functions that do not modify their list argument work on strings, but most aren't very useful.

Name	Arguments	Description	Example
`len`	1. A string.	Evaluates to the integer number of characters in the string.	`len('hello')` evaluates to `5`.

Formatting Methods

Name	Object and Arguments	Description	Example
`lower`	Object: String.	Evaluates to a copy of the string with all letters in lowercase.	If `s` stores `'Hello'`, `s.lower()` evaluates to `'hello'`.
`upper`	Object: String.	Evaluates to a copy of the string with all letters in uppercase.	If `s` stores `'Hello'`, `s.upper()` evaluates to `'HELLO'`.
`center`	Object: String. 1. An integer	Evaluates to a copy of the string that is padded evenly on both sides with spaces to be the length specified by the integer.	If `s` stores `'hi'`, `s.center(6)` evaluates to `' hi '`.
`center`	Object: String. 1. An integer 2. A character	Same as `center` but pads with copies of the character.	If `s` stores `'hi'`, `s.center(6, '*')` evaluates to `'**hi**'`.
`ljust`	Object: String. 1. An integer	Evaluates to a copy of the string that is padded on the right with spaces to be the length specified by the integer.	If `s` stores `'hi'`, `s.ljust(6)` evaluates to `'hi '`.
`ljust`	Object: String. 1. An integer 2. A character	Same as `ljust` but pads with copies of the character.	If `s` stores `'hi'`, `s.ljust(6, '*')` evaluates to `'hi****'`.
`rjust`	Object: String. 1. An integer	Evaluates to a copy of the string that is padded on the left with spaces to be the length specified by the integer.	If `s` stores `'hi'`, `s.rjust(6)` evaluates to `' hi'`.
`rjust`	Object: String. 1. An integer 2. A character	Same as `rjust` but pads with copies of the character.	If `s` stores `'hi'`, `s.rjust(6, '*')` evaluates to `'****hi'`.

Boolean Methods

Name	Object and Arguments	Description	Example
`isdigit`	Object: String.	Evaluates to the `True` if the string is entirely numbers or `False` otherwise.	If `s` stores `'12345'`, `s.isdigit()` evaluates to `True`.
`isalpha`	Object: String.	Evaluates to the `True` if the string is entirely letters or `False` otherwise.	If `s` stores `'Hello'`, `s.isalpha()` evaluates to `True`.
`isalnum`	Object: String.	Evaluates to the `True` if the string is entirely numbers and letters or `False` otherwise.	If `s` stores `'fn2187'`, `s.isalnum()` evaluates to `True`.
`islower`	Object: String.	Evaluates to the `True` if the string is entirely lowercase letters or `False` otherwise.	If `s` stores `'hello'`, `s.islower()` evaluates to `True`.
`isupper`	Object: String.	Evaluates to the `True` if the string is entirely uppercase letters or `False` otherwise.	If `s` stores `'HELLO'`, `s.isupper()` evaluates to `True`.
`startswith`	Object: String. 1. A string	Evaluates to the `True` if the object begins with the argument string and `False` otherwise.	If `s` stores `'hello'`, `s.startswith('hel')` evaluates to `True`.
`endswith`	Object: String. 1. A string	Evaluates to the `True` if the object ends with the argument string and `False` otherwise.	If `s` stores `'hello'`, `s.endswith('llo')` evaluates to `True`.

Search Methods

Name	Object and Arguments	Description	Example
`index`	Object: A string haystack. 1. A string needle.	Evaluates to the index at which the needle first appears in the haystack. If the needle does not appear, Python crashes.	If `s` stores `'banana'`, `s.index('ana')` evaluates to `1`.
`find`	Object: A string haystack. 1. A string needle.	Evaluates to the index at which the needle first appears in the haystack. If the needle does not appear, evalutes to `-1`.	If `s` stores `'banana'`, `s.index('ana')` evaluates to `1`.
`count`	Object: A string haystack. 1. A string needle.	Evaluates to the number of times the needle appears in the haystack.	If `s` stores `'banana'`, `s.count('na')` evaluates to `2`.

Manipulation Methods

Name	Object and Arguments	Description	Example
`strip`	Object: A string	A copy of the object with all whitespace removed from the left and right of the string.	If `s` stores `'\t hi\n '`, `s.strip()` evalutes to `'hi'`.
`strip`	Object: A string 1. A string	A copy of the object with all instances of any character in the argument removed from the left and right of the string.	If `s` stores `'-_-hi_-_'`, `s.strip('_-')` evalutes to `'hi'`.
`lstrip`	Object: A string (1. A string)	Same as `strip` but only on the left	
`rstrip`	Object: A string (1. A string)	Same as `strip` but only on the right	
`join`	Object: A string 1. A list of strings	Evaluates to the concatenation of all of the strings in the list with a copy of the object inserted between each pair of strings.	If `l` stores `['a', 'b', 'c']`, `'-'.join(l)` evaluates to `'a-b-c'`.
`split`	Object: A string	Evalutes to a list of strings derived from splitting the object up along all whitespace.	If `s` stores `'a \n b\t c'`, `s.split()` evaluates to `['a', 'b', 'c']`.
`split`	Object: A string 1. A string	Evalutes to a list of strings derived from splitting the object up along the argument string.	If `s` stores `'a-b-c'`, `s.split('-')` evaluates to `['a', 'b', 'c']`.
`replace`	Object: A string 1. A string to find 2. A string to replace	Evaluates to a copy of the object where all instances of the string to find have been replaced with copies of the string to replace.	If `s` stores `'a-b-c'`, `s.replace('-', '_')` evaluates to `'a_b_c'`.

Formatting

The format method operates on a string that contains placeholders like `{0}` or `{2}` (where 0 or 2 could be substituted for any integer). The format method evaluates to a copy of the object where each instance of `{n}` has been replaced with the n^{th} argument passed to format. For example:

>>> 'Happy {1}{2} birthday, {0}!'.format('Larry', 21, 'st')
'Happy 21st birthday, Larry!'

Saving a List of Strings to a File

A list of strings is typically stored to a text file by placing each string on its own line—that is, separating each string with a newline character. This only works when none of the strings contain a newline character. When the strings might

contain a newline character, you can usually employ the same strategy with a different character serving as a separator (such as a comma).

Writing a List of Strings to a File.

1. The list of strings is stored in the variable presidents.

   >>> presidents = ['washington', 'adams', 'jefferson']

2. Open the file where the strings are to be stored in write-mode.

   >>> fp = open('presidents.txt', 'w')

3. Join the list of strings with the newline character and write that string to the file.

   >>> fp.write('\n'.join(presidents))

4. Close the file.

   >>> fp.close()

Loading a List of Strings from a File. Read the contents of a file and split it along the newline operator.

```
>>> open('presidents.txt').read().split('\n')
['washington', 'adams', 'jefferson']
```

Keywords

Data sanitization—Removing the imperfections of user input so that it is ready for processing by the rest of your program. Example: removing extraneous whitespace at the beginning and end of an input string.

Escape—To strip a character of its special meaning in the context of a string. Example: within a string, the sequence `\'` creates a normal single quote character that does not start or end a string. Other escape sequences make use of the `\` charcter to insert characters that cannot be typed explicitly. Example: `\n` represents an empty line in a string.

Haystack—When searching for one item in a large database, the haystack is the database being searched.

Immutable—Data that cannot be modified. Immutability is often a useful guardrail to ensure that you don't accidentally modify something you shouldn't. Example: A string is immutable; Python will crash if you try to modify one of its characters as if it were a list.

Input validation—Ensuring that input provided by the user is correctly-formatted for use in a particular context. Example: making sure that the input for a dollar amount was actually a number.

Needle—When searching for one item in a large database, the needle is the item being searched for.

Raw string.—A string where no escaping takes place; \ characters are just \ characters. Example: preceding the opening quote of a Python string with r creates the raw string behavior.

Substring—A small piece of a larger string. Example: `'ell'` is a substring of `'hello'`.

Unicode—A standard for specifying all characters that can be displayed on a screen, including letters, numbers, special characters, whitespace, and even emojis.

Chapter 9

Iteration

In the past two chapters, we discussed how to create and modify lists, strings, and other list-like data. In the process, we introduced a vast array of methods, functions, and operators for manipulating this data. To this point, however, we have interacted with lists only in interactive Python.

In this chapter, we will bring lists and strings back to the world of full Python programs. To do so, we will need to revisit and augment our understanding of loops, which go hand-in-hand with list-like data. When working with lists, we are typically interested in performing an operation "for each element." For example, we might download each Supreme Court opinion, look for the keywords in each FCC comment, or calculate the bill for each client. When it comes to these sorts of tasks, loops and lists are inseparable.

We will not only find additional uses for while-loops, but also introduce a new kind of loop built specifically for lists—a for-loop. For-loops are designed to do the "for-each" style of iteration that lists typically demand. This pattern, in turn, will become a fundamental building block of most Python programs you will write going forward.

As you will quickly find, there are a few patterns that will surface over and over again whenever you mix lists with loops. To emphasize this theme, the final portion chapter is structured as a cookbook. Through a series of examples, we will describe recipes that capture the essence of list-centric Python programs. By the end of this chapter, you will be ready to leverage lists to their full potential in your Python programs.

9.1 For-Loops

9.1.1 Introducing Iteration

You will recall that we use loops whenever we need to repeat an operation more than once. For example, if we wanted to print a message to a user five times, we could use a while-loop as below:

```
1  # Create a counter variable.
2  count = 0
3
4  # Loop five times.
5  while count < 5:
6      print('Hello, user!')
7      count += 1
```

Line 2 of the program creates a counter variable that keeps track of the number of times we have printed the message. Initially, it hasn't been printed at all; in other words, it has been printed 0 times. On line 5, the while-loop itself begins. The body of the loop should repeatedly execute so long as the value of count is less than 5, the number of times we want the message to print. Inside the body of the loop, we print our desired message (line 6) and subsequently increase the value of count by 1. By now, the mechanics of this loop should be review.

Suppose that, rather than printing the same message five times, you had a list of messages, each of which you wanted to print once in order.

```
1  # A list of messages.
2  messages = ['Hello', 'Hi', 'Hola', 'Sup', 'Yo', 'Salutations']
```

Your program should be flexible enough that, even if the list of greetings were to change, getting longer or shorter, the code should continue to work properly. In other words, you can't hard-code the length of the list (6) into the loop. What should the corresponding while-loop look like? We will want to iterate through the items of the list one by one, starting with the first one. To do so, we will need a variable keeping track of the index of the list element we're currently printing.

```
3  # The current index of the list.
4  index = 0
```

Recall that the first item of a list is at position 0, so that is the value to which the variable index should be initialized. Our loop should continue iterating until the variable index reaches the end of the list. You will also recall that the last item in a list is at the position one smaller than the length of the list (position len(messages) − 1). This means that we should continue looping only so long as index is less than len(messages); when it reaches len(messages), the loop should exit or we risk going off the end of the list.

```
3  # The current index of the list.
4  index = 0
5
6  while index < len(messages):
```

We said earlier that we wanted the loop to print each item in the list one by one. To get each item in the list, we need to index the list messages with the current value of the variable index—in Python, messages[index]. We print this value and then increment index to move to the next item in the list.

```
3  # The current index of the list.
4  index = 0
5
6  while index < len(messages):
7      print(messages[index])
8      index += 1
```

On the first iteration of the loop, the value of index is 0, so we print 'Hello' (line 7) and increment index to 1 (line 8). Since index is still less than len(messages), the loop repeats, we print 'Hi' (line 7), and increment index to 2 (line 8). This process repeats again and again until index reaches 5, at which point we print the final item in the list, 'Salutations' (line 7), and increment index to 6 (line 8). When we perform the while-loop's boolean test, we find that it fails: index is equal to the length of the list. The loop exits and the program, having run out of code to execute, terminates. Overall, the output will look as follows:

```
Hello
Hi
Hola
Sup
Yo
Salutations
```

As the summary of the above program indicates, this process is effective but cumbersome. Every while-loop that iterates over a list will need several pieces of repetitive boilerplate. First, you will need to initialize an index variable to 0 (line 4). Then you will need to create a while-loop that iterates so long as the index variable is less than the length of the list (line 6). Finally, you will need to increment the index variable at the end of the loop body. Furthermore, every time you wish to refer to the current element of the list, you will need to use the square-bracket notation with the index variable inside (line 7).

9.1.2 Introducing For-Loops

Thankfully, Python (along with many other modern programming languages) recognizes that most loops are written to iterate over lists, so it includes a special kind of loop designed specifically for streamlining this process: a for-loop. We rewrite the previous program with a for-loop below.

```
1  # A list of messages.
2  messages = ['Hello', 'Hi', 'Hola', 'Sup', 'Yo', 'Salutations']
3
4  for message in messages:
5      print(message)
```

Six lines of boilerplate from the previous program have been replaced with two. More importantly, all of the index-manipulation complexity has disappeared completely. But how exactly does this for-loop work? As on line 4, every

for-loop begins with the keyword for (just as each while-loop begins with the keyword while). It then contains a new variable name (in this case, message) followed by the keyword in, a list (messages), and a colon. The for-loop will visit each element of the list messages. On the first iteration of the loop, the variable message will be set to the first element of messages ('Hello'). On the second iteration, message will be set to the second element of messages ('Hi'), and so on. On the final iteration, message will be set to the last element of messages ('Salutations').

In summary, a for-loop iterates over each element of a list (e.g., messages) in order. During each iteration, a variable (e.g., message) is set to the current element of the list. That variable can be used to refer to the current element of the list anywhere in the loop's body. In English, the first line of the for-loop can be read as, "For each `message` in the list called `messages`, perform the loop body."

In the program above, we could have chosen any valid variable name in place of message (the variable that iterates over each element of the list, also known as the for-loop's indexing variable).

```
1  # A list of messages.
2  messages = ['Hello', 'Hi', 'Hola', 'Sup', 'Yo', 'Salutations']
3
4  for item in messages:
5      print(item)
```

```
1  # A list of messages.
2  messages = ['Hello', 'Hi', 'Hola', 'Sup', 'Yo', 'Salutations']
3
4  for larry_the_lawyer in messages:
5      print(larry_the_lawyer)
```

Like for all variables, it is considered good style for the indexing variable to have a descriptive name (message rather than something generic like item or something nonsensical like larry_the_lawyer). Often, the descriptive names of a list and an indexing variable will go hand in hand. For example, the list will have a plural name (messages, clients, or court_cases) and the indexing variable a corresponding singular name (message, client, or court_case).

The list over which a for-loop iterates need not be a variable. It can be any expression that evaluates to a list. For example, we could iterate over the messages in alphabetical order using the sorted function as below:

```
1  # A list of messages.
2  messages = ['Hello', 'Hi', 'Hola', 'Sup', 'Yo', 'Salutations']
3
4  for message in sorted(messages):
5      print(message)
```

We could even drop a list straight into the for-loop:

```
1 for message in ['Hello', 'Hi', 'Hola', 'Sup', 'Salutations']:
2     print(message)
```

For-loops can be used to iterate over any list-like type, including tuples and strings. Iterating over a tuple will behave just like iterating over a list. Since strings are really just lists of characters, iterating over a string will visit each character one by one.

```
1 for letter in 'judicial':
2     print(letter)
```

The above program will print each letter in the string 'judicial' on its own line.

```
j
u
d
i
c
i
a
l
```

9.1.3 Nested For-Loops

For-loops can also iterate over nested lists, visiting one sub-list during each iteration. Consider the program below, which prints the length of each nested list within a larger list.

```
1 for sub in [[1, 2], [], [3, 4, 5, 6], [7], [], [8, 9]]:
2     print(len(sub))
```

On the first iteration, sub will be set to the first element of the for-loop's list: [1, 2]. The program will therefore print 2, the length of this list. On the second iteration it will be set to [], meaning the program will print 0. This will repeat for each of the remaining nested lists, printing 4, 7, 0, and 2.

What if we wanted to print the numbers themselves? We would need to iterate over the outer list, visiting each of the nested lists. We would then need to iterate over the elements of each nested list, printing the numbers one by one. This nested iteration will require nested for-loops, which are perfectly legal in Python.

```
1 for nested in [[1, 2], [], [3, 4, 5, 6], [7], [], [8, 9]]:
2     for number in nested:
3         print(number)
```

The variable nested will iterate through each of the nested lists. Initially, it will be set to [1, 2]. At this point, the inner for-loop will execute and the variable number will be set to the first element of nested, namely 1. Python will print 1 (line 3) and the inner loop will iterate again, setting number to 2.

After 2 is printed, the inner loop will run out of elements of nested, causing it to terminate.

The outer loop will then continue to its next iteration, setting nested to []. Since this list has no elements. the inner loop will never execute. The outer loop will iterate again, setting nested to [3, 4, 5, 6]. The inner loop will iterate through the elements of nested, setting number to 3, 4, 5, and 6 and printing them in order. This process will continue, with the outer and inner loops working in tandem to print the rest of the list. Eventually, we will see the numbers 1 through 9 printed in order, each on their own line.

Notice that the structure of the program mirrors the structure of the data. Iterating over lists nested within lists requires a for-loop nested within another for-loop.

9.1.4 For-Loops and Range

Often, you will find yourself wanting to iterate over a range of integers, which you can do with the range function. For instance, if you wanted to print the numbers 1 through 5, you could write the following program:

```
1 for number in range(1, 6):
2     print(number)
```

Remember that when we first saw the range function, we had to send it to the list () typecasting function to generate a list. This isn't necessary in a for-loop, which automatically treats the result of calling the range function as a list.

Recall that, when the range function has two arguments, it counts from the first argument up to *one below* the second argument. As expected, the program will produce the output below:

```
1
2
3
4
5
```

You can use this style of for-loop to even mimic the original while-loop formulation of our message printing program.

```
1 # A list of messages.
2 messages = ['Hello', 'Hi', 'Hola', 'Sup', 'Yo', 'Salutations']
3
4 for index in range(len(messages)):
5     print(messages[index])
```

The first two lines of the program have the list of messages as before. On line 4, notice that the for-loop is no longer iterating over the individual messages themselves; it's iterating over the *indices*. When provided with just one argument, the range function will produce a list of integers starting at 0 and stopping one before the value of its argument. The expression range(len(messages)) will

therefore produce the integers 0 through len(messages) − 1. In other words, index will iterate over all of the valid indices of the list messages. It will start at 0, then 1, then 2 and count all the way to the last index of the list. We can then use index just as we did in the while-loop case, accessing the current list element with the expression messages[index]. The only difference is that the for-loop will automatically increment the value of index for us.

When working with lists, we will generally rely on the for item in items: syntax rather than directly manipulating list indices with the range() function. It is generally better style to avoid list indices altogether whenever possible, and doing so also reduces the likelihood that you will accidentally introduce a bug or create a subtle off-by-one error. Still, it is useful to know the difference between for item in items:, which directly iterates over each item in the list, and for index in range(len(items)):, which iterates over the list's indices.

9.2 While-Loops vs. For-Loops

In the previous section, we introduced a new kind of loop that is specially designed to iterate over lists. These for-loops have clear benefits: they are often manipulate lists in a clearer, more succinct fashion than while-loops. Yet we also saw that several of the tasks we confronted could be solved with either kind of loop, and the benefits of choosing one over the other were not always clear. When is it better to use which kinds of loops? Are there some problems that are impossible to solve with one or the other? In this section, we delve into these questions, formally laying out your capabilities and choices as a programmer.

9.2.1 All For-Loops are While-Loops

It turns out that all for-loops can be rewritten as while-loops. A for-loop can be thought of as a while-loop that takes care of some extra bookkeeping on your behalf. If we wanted to iterate over a list with a while-loop, we would need to create an index variable to keep track of the current element of the list, increment it at each loop iteration, and ensure that the loop exits when we have no list elements left. A for-loop does all of this work automatically, meaning that any for-loop can be converted into a while-loop by making these operations explicit. For example, the following for-loop prints every element in a list of strings.

```
1 names = ['George', 'John',  'Thomas', 'James', 'Andrew']
2
3 for name in names:
4     print(name)
```

To convert this for-loop into a while-loop, we need to create a variable that explicitly keeps track of the current index. It must be incremented at the end of the loop body. The while-loop's boolean condition should be that this index remains smaller than the length of the list. Finally, whenever we access the

current element of the list, we will need to use standard list indexing with square brackets.

```
1 names = ['George', 'John',  'Thomas', 'James', 'Andrew']
2
3 index = 0
4 while index < len(names):
5     print(names[index])
6     index += 1
```

It turns out that this recipe is sufficient to convert any for-loop into a while-loop. We can conclude that the decision to use a for-loop over a while-loop is always optional. In other words, whenever we use a for-loop, there is always a corresponding while-loop we might have chosen instead. For-loops are a convenience offered by Python, but they are not mandatory.

9.2.2 Not All While-Loops are For-Loops

However, the converse is not true. There are some while-loops that would be impossible to write as for-loops. In Python, a for-loop will always iterate a limited number of times; it will loop once for each of the items in the list over which it is iterating. If a for-loop is provided a list with a thousand items, it will iterate a thousand times; if the list has only two items, it will iterate twice. While loops, as we discovered several chapters ago, do not have this limitation. In fact, we have built many while-loops that, in theory, might have continued iterating forever if a user continued providing input. For the simplest example of a while-loop that can't be written as a for-loop, consider the following:

```
1 while True:
2     print('Infinite  loop!')
```

You might not want to write Python programs like this very often, but we have already created useful programs that are only minor variations on the one above. Consider, for example, a program that continues reading user input so long as the user continues to provide it. The program cannot know in advance how much input the user will provide, and the user could continue supplying input forever. The for-loops we have discussed are not capable of facilitating these sorts of programs.

9.2.3 Choosing a Loop

As we have seen in the preceding sections, some situations will leave you no choice; you will have to use a while-loop in many contexts where a for-loop cannot facilitate the program you need to write. As with variable names and many other decisions in your Python programs, the choice between while-loops and for-loops is often a judgment call left to you as a programmer. You should strive to ensure your code is concise, readable, and easy to change later.

In general, programs that iterate over lists are better framed as for-loops. With that said, for-loops suffer from two key limitations that will often force you to use while-loops instead. First, they can only visit each list element in order from first to last and only once. Second, the for-loop body does not have access to the index of the current element. If you are considering a list where knowing the list index matters, for example where every other element of the list should be treated differently, a for-loop will not have enough information to convey your desired instructions to Python.

However, Python does have one convenient tool that can give for-loops this power: the enumerate function. The enumerate function takes one argument, a list. It evaluates to a list of the same length as its argument, with a key modification: each list element is replaced with a tuple containing the element's index and the element. Consider the examples below:

```
>>> enumerate(['Alice', 'Bob', 'Charlie'])
[(0, Alice), (1, Bob), (2, Charlie)]
>>> enumerate([99, 32, 61, 78, 84])
[(0, 99), (1, 32), (2, 61), (3, 78), (4, 84)]
>>> enumerate([True])
[(0, True)]
```

In the first example, the element at index 0, `'Alice'`, is replaced by the tuple `(0, 'Alice')`. Likewise, the elements at positions 1 and 2 are replaced with the tuples `(1, 'Bob')` and `(2, 'Charlie')`. Each of the other examples is handled in a similar fashion.

When you want a for-loop to have access to the list indices in addition to the list elements, you can iterate over the list passed to the enumerate function. Up to now, every for-loop we have seen has created only a single piece of data on each iteration, which we have stored in a single variable. A list passed to the enumerate function creates *two* pieces of data on each iteration—the two items in the tuple (the index and the element at that index). To receive these two elements, you can list two variables after for keyword, separating the variable names with a comma.

```
1  # A list of messages.
2  messages = ['Hello', 'Hi', 'Hola', 'Sup', 'Yo', 'Salutations']
3
4  for index, message in enumerate(messages):
5      print('The greeting at index {0} is {1}'.format(index, message))
```

On line 4, notice that the for-loop creates two variables, index and message. These variables correspond to the two elements of each tuple created by the call to the enumerate function. On each iteration, Python retrieves the next tuple in the list, unpacks it into two separate elements, and saves those elements to the two variables. (If the list contained tuples with more than two elements, you would need a correspondingly larger number of variables to receive the elements of the tuples.)

When we run the program, we see the following output, confirming that we were able to get access to the element indices.

```
The greeting at index 0 is Hello
The greeting at index 1 is Hi
The greeting at index 2 is Hola
The greeting at index 3 is Sup
The greeting at index 4 is Yo
The greeting at index 5 is Salutations
```

9.3 For-Loop Recipes

Now that you are familiar with the relationship between while-loops, for-loops, and lists, we will walk you through strategies for managing many of the most common scenarios that you will encounter in practice. This section is structured as a cookbook, providing you with recipes and patterns that you can apply directly in your programs.

9.3.1 Printing

The most basic design pattern is simply iterating over a list and printing the elements. This has been the primary running example throughout the chapter, so we note its existence but do not repeat its analysis.

9.3.2 Summarizing

Few programs simply print the elements in a list. Instead, they aim to extract some sort of information from a list that can be used for later computation. A simple example might be a program that adds up a list of integers or finds the longest word in a list of strings. These programs will need to visit each element of the list and update some running total.

Consider the program below, which adds a list of integers.

```
1  # The list of integers.
2  numbers = [29, 40, 42, 19, 97, 79, 61]
3
4  # The running total.
5  total = 0
6
7  # Add the numbers.
8  for number in numbers:
9      total += number
10
11 # Print the total.
12 print('The total is {0}'.format(total))
```

The program begins as usual: we create a list of integers on line 2. In reality, we would likely read this list as input, but this example has simplified that part to draw attention to the pieces that matter for this chapter's focus. On line 5, we create a variable that will serve as our running total of the elements in the list. It is initially set to 0, as we haven't yet examined any of the list elements. The for-loop itself (line 8) iterates through each number in the list, adding it to the running total (line 9). When the loop exits, total should have summed every element in the list, meaning that we can print it on line 12.

We can use a similar pattern to find the longest word in a list of strings. Rather than keeping track of a running total, we need to keep track of the longest word we've seen so far.

```
1  # The list of words.
2  words = ['apple', 'banana', 'watermelon', 'cantaloupe', ...]
3
4  # The longest word so far.
5  longest = ''
6
7  # Find the longest word.
8  for word in words:
9      if len(word) > len(longest):
10         longest = word
11
12 # Print the longest word.
13 print('The longest word is {0}'.format(longest))
```

This program looks superficially similar to the previous program. On line 2, it creates a list of strings; we leave an ellipses where many more words could be added. (The ellipses are there for display and won't work in Python, of course.) Just as before, we initialize a variable that keeps track of our running summary of the list (line 5). This time, rather than adding all of the elements of the list together, we are keeping track of just one list element: the longest element we have seen. Before the loop starts, we haven't yet seen any words, so we need to set longest to some default value that will assuredly be overwritten by the first word we see. One possibility is to set it to the None value and add an if-statement to overwrite it on the first iteration of the for-loop on line 8. Alternatively, we could set longest to the empty string, which will be shorter than any word the program sees. Within the for-loop (line 8) is an if-statement (line 9) that checks whether each word that we visit is longer than the current value of longest; if it is, the program overwrites longest with the current word, ensuring longest stays up to date. After the loop ends, we know we have seen every word, so we must have the longest word in the entire list.

This program can be thought of as a sort of tournament: at every loop iteration, the next word in the list faces off against the current champion, with the winner moving on to the next round. By the end, we have found the overall winner.

9.3.3 Transforming

So far, all of the programs we have discussed have been passive observers, looking at lists and gleaning information without making changes. In many cases, we wish to modify each list element in the same way. For example, we might wish to divide each element in an integer list by 100 or concatenate the prefix 'Justice ' to a list of names of Supreme Court justices.

Unfortunately, this is a job for which for-loops are less handy. Consider our first try at the program that converts a Supreme Court justice's name into his or her formal title.

First, we create a list containing all of the justices on the Supreme Court. Although the Justices listed have just two names (e.g., 'Elena Kagan'), some justices might have several names (e.g., 'Ruth Bader Ginsburg').

```
1  # The supreme court justices.
2  justices = ['Elena Kagan', 'John Roberts', ...]
```

Then we begin the loop itself.

```
3  # Convert each justice's name.
4  for justice in justices:
5      # Split the first, [middle,] and last names.
6      names = justice.split()
7
8      # Create the justice's title.
9      last_name = names[-1]
10     title = 'Justice ' + last_name
11
12     # Save the name back to the list.
13     ???
```

We iterate over the justices on line 5. One by one, the names of the justices are saved to the variable justice and processed by the loop. First, the name is split into a list of names on line 7. For example, Ruth Bader Ginsburg is split into the list ['Ruth', 'Bader', 'Ginsburg']. On line 10, we select the last element of this list, which is the justice's last name. The prefix 'Justice ' is concatenated in front of this last name, creating the justice's official title on line 11. On line 14, we'll need to save the value of the variable title back to the list somehow; we will discuss how to do so in more detail in a moment.

```
16 # Print the names.
17 for justice in justices:
18     print(justice)
```

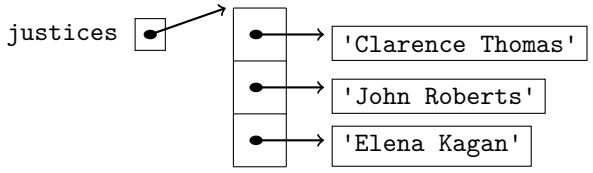
Finally, after the for-loop that transforms the list ends, we print the updated version of the list stored in justices, which should print strings like 'Justice Kagan' and 'Justice Ginsburg'.

The primary challenge of this example is filling in line 14, which aims to update the current element of the list with the value stored in the variable title. Initially, we might assume that we can simply save the value of title back to

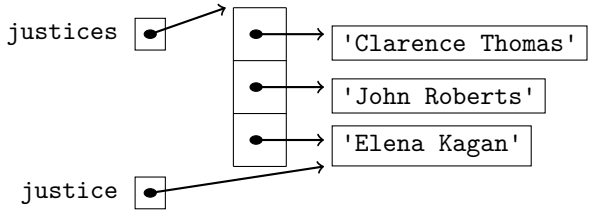
variable justice, which stores the current value of the list. In other words, by this logic, line 14 should read:

```
14      justice = title
```

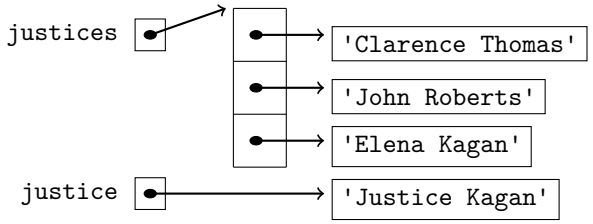
Unfortunately, overwriting the value of justice does not overwrite the corresponding element in the original list (justices). To understand exactly why this assignment fails, we will need to learn more about what exactly Python does when it evaluates a for-loop. Recall that the justices variable points to a list in memory. This list, in turn, points to the strings representing the names of the Supreme Court justices.



When the for-loop iterates through the list, the justice variable is assigned to each of these strings in turn.



When we assign to justice, we are changing only the value that justice points to, not the item in the list.



If we want to edit the list itself, we will need to use the list indices to modify the list directly. At this point we have two choices: we can either move to a while loop or use a for-loop with the enumerate function. Neither choice is ideal. If we switch to a while-loop, we will have to manage the list indices directly.

```
3  # Convert each justice's name.
4  index = 0
5  while index < len(justices):
6      # Split the first, [middle,] and last names.
7      names = justices[index].split()
8
9      # Create the justice's title.
```

```
10      last = names[-1]
11      justices[index] = 'Justice ' + last
12
13      index += 1
14
15  # Print the names.
16  for justice in justices:
17      print(justice)
```

Since we are using a while loop, we must create a variable that keeps track of the current list index (line 4) and set a standard boolean condition for the loop (line 5). Every time we refer to the current list item, we need to say `justices[index]` rather than simply `justice` as in the for-loop case. Finally, we must also increment the value of index (line 13). This formulation of the program does accomplish our goals, modifying each element of the list from `'Elena Kagan'` to `'Justice Kagan'`, but it sacrifices the conveniences of for-loops.

We could also try reformulating this program using a for-loop with the enumerate function.

```
3   # Convert each justice's name.
4   for index, justice in enumerate(justices):
5       # Split the first, [middle,] and last names.
6       names = justice.split()
7
8       # Create the justice's title.
9       last = names[-1]
10      justices[index] = 'Justice ' + last
11
12  # Print the names.
13  for justice in justices:
14      print(justice)
```

Rather than just extracting the name of the justice, the for-loop also extracts that justice's index in the list `justices` (line 4). This means that, on line 10, we can assign back to the original list, ensuring that it gets updated with the justice's title.

This version of the program enjoys some of the brevity that for-loops enable: we can refer to the current element of the list as simply `justice` on line 6 and avoid some manipulation of list indices. The only place where a list index appears is on line 10, when we are assigning back to the original list. The drawback of this approach is that we have to mix two different ways of referring to the current element of the list. When we read the element, we can call it `justice`, but when we overwrite it (line 10), we have to refer to it as `justices[index]`. To a reader looking at this code for the first time, these two names will prove quite confusing. To be consistent, we could refer to it as `justices[index]` everywhere, but doing so is tantamount to using a while-loop. Neither of these choices is perfect, and you—as a programmer—will have to select the appropriate alternative for each context.

We hope that this discussion of style has not distracted from the core lesson of this section: a recipe for iterating over a list and modifying each element in place.

9.3.4 Generating Lists

In the previous section, we discussed techniques to modify each element of a list. For example, we might wish to convert a list grades in decimal form ([.88, .92, .75]) into a list of strings containing those numbers written as percentage scores (['88%', '92%', '75%']). Many times, you will wish to perform this sort of transformation without modifying (and thereby destroying) the original list. Instead, you can create a new list that contains the modified values, leaving the original list untouched.

```
1  # The list of numerical grades.
2  grades = [.88, .92, .75]
3
4  # Create a list of string scores.
5  scores = []
6
7  # Convert numerical grades into string scores.
8  for grade in grades:
9      score = str(grade * 100) + '%'
10     scores.append(score)
11
12 # Print the scores.
13 for score in scores:
14     print(score)
```

Superficially, this program is similar to examples from the preceding section. We begin with an initial list (line 2) that we wish to transform. Rather than modifying the old list, grades, we create a new empty list, scores (line 5), that will store our ouput. The for-loop (lines 8-10) iterates over each grade and converts it into a string score (line 9). It stores this result by appending it onto the end of the new list, scores (line 10). In doing so, it avoids modifying grades. Each entry of grades will have a corresponding entry at the same index in scores. Since we are building scores in the same order as we are iterating over grades, the resulting lists will be in the same order.

Notice that the code above is simpler and a bit easier to understand than the examples that tried to modify a list rather than generate a new one. For one thing, the new code does not require the storage or handling of indices, which tend to be a bit harder to track than list items themselves. Also, because the scores list is separate and apart from the grades list, it is easier to keep them separate and distinct in your mind. For this reason, you will see this pattern a lot in this book: a new list will be initialized with the empty list, [] , and constructed using a for-loop, one item at a time, based on transformations of the items in an old list.

One other benefit of building a new list is that it need not contain every element that appeared in the original list. Consider a slight modification to the previous program.

```
1  # The list of students and their numerical grades.
2  grades = [('Larry', .88), ('Barry', .75), ('Sherry', .92)]
```

This time, our list of grades is really a list of tuples. Each tuple contains two pieces of information: the name of a student and the grade she or he received. For example, Larry received an 88%. This time around, we wish to make a list of all of the students who passed the exam. We define passing to be a grade of 80% or higher. Just as before, we will need to create an initially-empty list that will eventually contain the names of the students who passed the exam.

```
4  # Create a list of students who passed the exam.
5  passing = []
```

This time, the for-loop creates two variables on each iteration (line 8). The student variable stores the name of each student (the first element of the tuple). The grade variable stores the decimal number containing the corresponding student's grade.

```
7  # Convert numerical grades into string scores.
8  for student, grade in grades:
9      if grade >= .8:
10         passing.append(student)
```

The for-loop's body checks whether the grade is greater than or equal to .8, which is the lowest possible passing grade. If it is indeed a passing grade, the student's name is added to the passing list. When the loop is done executing, passing will contain all of the names of the passing students. In this case, it would store 'Larry' and 'Sherry' but not 'Barry' since he did not pass the exam. Finally, the list of passing students is printed.

```
12 # Print the scores.
13 for passing_student in passing:
14     print(passing_student)
```

This kind of program is said to filter the original list since it applies a test to each element and adds only those that pass to the resulting list.

9.3.5 Generating a List from Input

You will often wish to build a list from user input. In previous chapters, we have written programs that collect user input so long as the user continues to enter information. For example, we wrote programs that calculated legal bills using the time intervals that an attorney had spent working for a client. In these programs, we first asked for the name of a client and then collected the lengths of each time interval spent working on the client's case. The program then computed the overall bill for the client before requesting the name of another

client. In the past, we had no way to store the potentially lengthy lists of client names and time intervals; instead, we had to add time intervals together as we read them and output one bill before starting on another. Now that we have lists, we can save this information to variables.

```
1  # Store the client names and time intervals.
2  clients = []
3  intervals = []
```

This program builds two lists. The first is a list of client names as strings (clients on line 2). The second is a list of lists of floating point numbers containing each individual time interval that the attorney spent working for the client (intervals on line 3). The variable intervals is a list of lists because each entry corresponds to the list of intervals during which the attorney worked for the client. For example if clients were ['Larry', 'Mary'], intervals might be [[1, 3, 2.5], [.5, .75]], indicating that the attorney worked intervals of 1, 3, and 2.5 hours for Larry and .5 and .75 hours for Mary.

```
5  # Read the first client.
6  client = input('Client: ')
7
8  while len(client) > 0:
9      clients.append(client)
```

On line 6, the program reads the name of the first client. If the name is not blank (the while-loop boolean test on line 8), the new client's name is appended to the end of clients.

```
10     current_intervals = []
11
12     # Read the first time increment for this client.
13     interval = input('Hours working on client: ')
14
15     # Read subsequent time increments.
16     while len(interval) > 0:
17         current_intervals.append(float(interval))
18         interval = input('Hours working on client: ')
```

Next, in line 10, a temporary list current_intervals is created to gather intervals for this client. This list is initially empty, indicating that we currently have recorded no time intervals for that client. On line 13, we read the first of these time intervals. The loop on lines 16-18 reads successive time intervals so long as they are not empty. Each interval is appended to current_intervals. When the user provides a blank entry for an interval, signaling all of the intervals have been entered, current_intervals will contain all of the intervals for the current client.

```
20     # Store the current client's intervals in the larger list.
21     intervals.append(current_intervals)
```

Now that all of the intervals for the current client have been stored in current_intervals, we can append this list of intervals to intervals, which is the list of lists of intervals that keeps track of the intervals for all clients.

```
23      # Read the next client.
24      client = input('Client: ')
```

Finally, we read the next client's name on line 23, causing the entire process to repeat.

This program demonstrates the process of building a list using user input. Initially, empty lists are created (lines 2-3). The user input (lines 6, 13, 18, and 24) is appended onto the ends of these lists (lines 9, 17, and 21), slowly building the lists we can use for later computation. The code above never uses the lists for any other purpose, but we could imagine writing additional lines of code that calculate client-by-client bills or perform other sorts of analysis on the complete billing data.

9.3.6 Parallel Lists

In all of the examples we have discussed so far, programs have been provided with just one list as input. In many cases, you will work with data that is split across several parallel lists. Consider the following example: you intend to examine several thousand public comments on the draft of a rule proposed by an important government agency. These comments are provided to you in the form of several lists. One list, authors, contains the names of the authors of each comment. Another, dates, contains the date and time at which the comment was filed. Yet another, emails, stores the authors' email addresses. You could probably imagine a dozen other lists of this kind of metadata about comments. Most important is comments, which contains the comments themselves. These lists are in parallel, meaning that the elements at a given list index all go together. For example, authors[12] is the author of comments[12], which was submitted from emails[12] at on dates[12]. The same is true for any single index. Since the elements of the lists are grouped in this way, all lists should have the same length.

A normal for-loop is designed to handle only one list at a time, so it cannot iterate over all of these lists simultaneously. One alternative would be to use a while-loop and keep track of the index of the current element explicitly.

```
1  # Initialize variables.
2  index = 0
3  form = 'The comment by {0} was {1} words long.'
4
5  while index < len(comments):
6      # Compute the number of words in the comment.
7      words_in_comment = len(comments[index].split())
8
9      # Print information about the comment.
```

```
10      print(form.format(authors[index], words_in_comment))
11
12      # Increase the index for the next iteration.
13      index += 1
```

The program uses a standard while-loop to iterate over the parallel lists. The index variable, index, is initialized to 0 on line 2, tested in the loop condition on line 5, and incremented at the end of the loop on line 13. Line 10 prints the author of the comment and the number of words in the comment (calculated on line 7 by splitting it into words and then counting the length of the resulting list). This approach successfully iterates over all of the lists in parallel by sharing a single index variable; we could imagine adding several more parallel lists without altering our fundamental approach. One downside to this strategy, however, is that it forces us to explicitly deal with list indices, ugly bookkeeping that for-loops are designed to hide away.

It turns out that we can, indeed, use for-loops for iterating over parallel lists with the help of the zip function. The zip takes as its arguments a series of lists of the same length. It "zips" these lists together into a single list of tuples, where each element of the new list is a tuple of the elements of the argument lists at that index. For example, if we had the parallel lists

```
fruits = ['apple', 'banana', 'grape']
```

and

```
colors = ['red', 'yellow', 'purple']
```

the expression `zip(fruits, colors)` would evaluate to a single list

```
[('apple', 'red'), ('banana', 'yellow'), ('grape', 'purple')]
```

We can now successfully iterate over this zipped list with a for-loop. Rewriting the program above:

```
1  form = 'The comment by {0} was {1} words long.'
2
3  for comment, author in zip(comments, authors):
4      # Compute the number of words in the comment.
5      words_in_comment = len(comment.split())
6
7      # Print information about the comment.
8      print(form.format(authors, words_in_comment))
```

By zipping the lists on line 3, we create a list of tuples. Each for-loop iteration retrieves the next tuple in the zipped lists, assigning the first element of the tuple to the variable comment and the second to author (line 3). Within the loop, we can directly refer to these variables, obviating any need to do indexing. Otherwise, the body of the loop remains unchanged.

You can iterate over additional parallel lists by providing them as additional arguments to zip and, correspondingly, creating more variable names to the left

of the in keyword. For example, we could add the date that the comment was made.

```
1 form = 'The comment by {0} on {1} was {2} words long.'
2
3 for comment, day, author in zip(comments, dates, authors):
4     # Compute the number of words in the comment.
5     words_in_comment = len(comment.split())
6
7     # Print information about the comment.
8     print(form.format(authors, day, words_in_comment))
```

9.3.7 Nested Iteration

Just as we encountered nested while loops, it is sometimes necessary to nest iteration—to iterate over data on each iteration of a larger loop. Often, this need is dictated by the data. A list containing nested lists usually requires iteration containing nested iteration. The first iteration will step through the outer list, assigning each inner list to a variable. Then, a second level of iteration will iterate through the values stored in the inner lists. In other cases, you will simply need to iterate over the same list twice at different levels of nesting.

To make this discussion concrete, imagine that you have a list of your friends and their birthdays. You are interested in finding the two friends whose birthdays are closest to one another (since you could save time and energy by hosting a joint birthday party). As shorthand, we will call these two people "birthday partners." To make this example concrete, your friends' names are stored as a list of strings.

```
1 # List of friends.
2 friends = ['Larry', 'Sherry', 'Gary', 'Mary', 'Barry', 'Kary']
```

In a separate list are their birthdays. Each birthday is a number from 1 (representing January 1st) to 366 (representing December 31st)—we count to 366 since we reserve a number for February 29th. This parallel list contains your friends' birthdays in the same order as before. In other words, index 2 of the friends list contains Gary's name, and index 2 of the birthdays list contains Gary's birthday.

```
4 # List of birthdays.
5 birthdays = [235, 300, 128, 209, 36, 194]
```

To find the two friends who have the nearest birthdays, we will have to consider all possible pairs of friends (Larry and Sherry, Larry and Gary, Sherry and Gary, etc.). Whichever of these pairs has birthdays that are closest to one another will be our birthday partners. To implement this strategy in Python, we will need two nested loops. The outer loop will iterate over each friend one by one. For each friend, the inner loop will iterate over all of the other friends. In this way, we will consider every single possible pair of friends. In our list above, for

example, you will start by comparing other friends' birthdays to Larry's—Larry to Sherry, then Larry to Gary, then Larry to Mary, etc. You will then move on to Sherry, comparing her to Larry, Gary, Mary, etc.

Now, let's see how the strategy described above works in code—it will probably be easier to understand in Python than in English. First, we will need to create some variables to keep track of the best pair of people we have seen so far. We will update these variables every time we find two people whose birthdays are closer than the current best.

```
7  # The best pair of birthday partners so far.
8  best_days_apart = 183 + 1
9  best_person1 = ''
10 best_person2 = ''
```

The variable best_days_apart keeps track of how many days apart the current best pair of birthday partners are. We initialize it to half a year plus one day (183 + 1), since no two birthdays can possibly be more than than half a year apart. Remember that the shortest distance between two birthdays may wrap around through the end of the year; two people born on December 31st (366) and January 1st (1) are only 1 day apart, not 365. This means that the first pair of people we examine, no matter how distant their birthdays, will always be less than 184 days apart. We initialize the names of the best pair of people to the empty string, since we will overwrite these values at the first actual pairing we consider. (We could alternatively initialize these names to None.)

Next, we create a for-loop that iterates over each person.

```
12 # Examine each person...
13 for friend1, bday1 in zip(friends, birthdays):
14     # Statements inside the loop.
```

Since friends and birthdays are parallel lists, we jointly iterate over them using the zip function. The variable friend1 marks the first friend in the pair that we are considering; bday1 stores that friend's birthday. Each of these variables has the number 1 at the end since we will be considering a second friend in the nested loop.

How do we fill in the loop? There will need to be an inner loop that allows us to consider pairs of friends.

```
12 # Examine each person...
13 for friend1, bday1 in zip(friends, birthdays):
14
15     # ...against every other person.
16     for friend2, bday2 in zip(friends, birthdays):
17         # If this pair is better than our best pair, update
18         # our best pair accordingly.
```

The inner loop gets its own index variables: friend2 also iterates over all friends, and bday2 stores friend2's birthday.

Finally, we can fill in the inner loop by checking whether the two friends have closer birthdays than the closest pair yet considered.

```
12 # Examine each person...
13 for friend1, bday1 in zip(friends, birthdays):
14
15     # ...against every other person.
16     for friend2, bday2 in zip(friends, birthdays):
17         # Calculate the number of days between birthdays.
18         later = max(bday1, bday2)
19         earlier = min(bday1, bday2)
20         days_apart = later - earlier
21
22         # If days_apart is bigger than half a year, this will
23         # use the other way around the calendar
24         days_apart = min(days_apart, 366 - days_apart)
25
26         # Update our best pair.
27         if days_apart < best_days_apart:
28                 best_days_apart = days_apart
29                 best_person1 = friend1
30                 best_person2 = friend2
```

The body of the loop might look a little cryptic at first glance. Lines 18 and 19 determine which birthday falls later in the calendar and which birthday falls earlier in the calendar. This means that, when we calculate the number of days between the two birthdays on line 20, we always end up with a positive number.[1]

To finish calculating the number of days between the birthdays, we need to avoid one other pitfall. If bday1 was January 1st (1) and bday2 was December 31st (366, line 20 would calculate that the birthdays were 365 days apart, when, in reality, they're just one day apart. In other words, we need to account for the fact that the calendar is circular. Line 20 only calculates the number of days between the birthdays moving forward in time from the earlier birthday to the later birthday. We also need to consider the days apart when we count backwards from the later birthday to the earlier birthday. We can accomplish this by noticing that, if two birthdays are x days apart moving forward in time, they are 366 − x days apart moving backwards in time. That is, the amounts of time moving forward and backwards should add up to one full year (i.e., 366 days). The example birthdays before are 365 days apart counting forwards and 1 day apart counting backwards. Line 24 puts this idea into code, setting days_apart to whichever of the forward and backward differences is smaller.

[1] Consider what would happen if we didn't take this step. If we tried to compute the days apart with the calculation bday1 − bday2, we'd get a negative number of days apart every time bday1 fell earlier in the calendar. The reverse would happen if we tried to compute the days apart with the expression bday2 − bday1. Lines 18-20 ensure that the result is always positive.

Finally, lines 28-30 update the variables that store the friends with the closest pair of birthdays we've seen so far. In other words, if days_apart is smaller than the best days apart we've seen so far, we update overwrite the previous closest pair of friends with the one we're currently examining.

Finally, the last two lines of the program print the closest pair of birthdays.

```
32 # Print the result.
33 print('{0} and {1}'.format(best_person1, best_person2))
34 print('Days apart: {0}'.format(best_days_apart))
```

When we try to run this program, we'll immediately see a key problem with our approach.

```
$ python3 birthdays.py
Larry and Larry
Days apart: 0
```

This result says that the two people with the closest birthdays are Larry and himself! This output is true but unhelpful. Looking back, the program did exactly as we asked. It compared all possible pairs of people and found the two with the closest birthdays. When friend1 was 'Larry' and friend2 was 'Larry', the program found that the birthdays were 0 days apart. Since it couldn't find a better answer afterwards, that was the result it printed. The key problem is that the program we wrote wasn't the program we intended to write. We wanted to only compare pairs of people who were *different*. In other words, whenever friend1 and friend2 refer to the same person, we should skip that pair. We can encode that logic by opening the nested loop with an if-statement.

```
12 # Examine each person...
13 for friend1, bday1 in zip(friends, birthdays):
14
15     # ...against every other person.
16     for friend2, bday2 in zip(friends, birthdays):
17         # Skip pairs that refer to the same person.
18         if friend1 == friend2:
19             continue
20
21         # Calculate the number of days between birthdays.
22         later = max(bday1, bday2)
23         earlier = min(bday1, bday2)
24         days_apart = later - earlier
25
26         # If days_apart is bigger than half a year, this will
27         # use the other way around the calendar
28         days_apart = min(days_apart, 366 - days_apart)
29
30         # Update our best pair.
31         if days_apart < best_days_apart:
```

```
32                  best_days_apart = days_apart
33                  best_person1 = friend1
34                  best_person2 = friend2
```

Lines 18 contains an if-statement that checks whether friend1 and friend2 contain the same person. If they do, then the continue statement on line 19 will skip to the next iteration of the nested for-loop (line 16). In doing so, it will avoid all of the updating logic on lines 31-34. When we run the program, we can confirm that it produce the correct result.

```
$ python3 birthdays.py
Mary and Kary
Days apart: 15
```

Style note. Throughout the birthday program, we mentioned special values like 366 (the number of days in the year) and 183 (the number of days in half a year. Although the meaning of these magic numbers may seem obvious now, they will appear cryptic to future readers. In order to make the meaning of these magic numbers more obvious, it is considered good style to save them to constant variables (shorthand: constants) at the beginning of the program.

```
1  DAYS_IN_YEAR = 366
2  HALF_A_YEAR = 183
```

Constants are just normal variables, but—stylistically—they store values that will never change, like the number of days in the year or the number of feet in a mile. To distinguish them from normal variables, constants are named using all capital letters. Constants are valuable for three reasons.

1. Costants document what value a magic number represents. The statement that DAYS_IN_YEAR = 366 clearly indicates the significance of the number 366.

2. Rather than using the magic number later in the program itself, we can use the name of the constant, making subsequent computation easier to follow.

3. Constants centralize all of the magic numbers in one place, meaning they're easy to update later if they need to be changed. For example, consider a program that assumed that there were nine Supreme Court justices. If the program were poorly-written, the number 9 would appear in expressions throughout the program without being saved to a constant. In 2016, when there was an extended period of time during which there were only eight justices, all of these appearances of the number 9 would need to be changed. Not only would this task be tedious, but it would introduce the risk of missing a few spots and creating a buggy program. If, instead, the program had a constant named NUMBER_OF_JUSTICES that was used throughout the program, changing the number of justices would be easy.

Going back to the task at hand, we could update the rest of the program to make use of the constants.

```
3  # List of friends.
4  friends = ['Larry', 'Sherry', 'Gary', 'Mary', 'Barry', 'Kary']
5
6  # List of birthdays.
7  birthdays = [235, 300, 128, 209, 36, 194]
8
9  # The best pair of birthday partners so far.
10 best_days_apart = HALF_A_YEAR + 1
11 best_person1 = ''
12 best_person2 = ''
13
14 # Examine each person...
15 for friend1, bday1 in zip(friends, birthdays):
16
17     # ...against every other person.
18     for friend2, bday2 in zip(friends, birthdays):
19         # Skip pairs that refer to the same person.
20         if friend1 == friend2:
21             continue
22
23         # Calculate the number of days between birthdays.
24         later = max(bday1, bday2)
25         earlier = min(bday1, bday2)
26         days_apart = later - earlier
27
28         # If days_apart is bigger than half a year, this
29         # use the other way around the calendar
30         days_apart = min(days_apart, DAYS_IN_YEAR - days_apart)
31
32         # Update our best pair.
33         if days_apart < best_days_apart:
34                 best_days_apart = days_apart
35                 best_person1 = friend1
36                 best_person2 = friend2
```

9.4 Conclusion

In this chapter, we connected several disparate threads together into one coherent whole. We spent the last two chapters delving into the particulars of lists and other list-like types. Three chapters ago, we discussed a new control-flow mechanism: loops. Here, we unified these components into constellation of design patterns that will form the backbone of most Python programs that you

will write. Lists—collections of data—are an essential ingredient in even the most sophisticated Python programs. For-loops give you the power to create and process them

Now that you have the power to use lists and loops in tandem, we will explore one final way of organizing data, dictionaries. Where lists organize information by keeping it in a particular order, dictionaries dispense with order in favor of other criteria for indexing data. With dictionaries, our exploration of Python data will be complete, allowing us to move into higher-level ways of organizing programs (functions and libraries) and storing data (files and directories).

9.5 Cheatsheet

Iterating with While-Loops

We want to perform an action on every item in a list called items.

```
1 index = 0
2 while index < len(items):
3     # Perform the desired operation on items[index].
4     index += 1
```

On line 1, we create a variable that keeps track of the index of the list element we plan to operate on during the current loop iteration. Initially, we operate on the first index of the list—index 0.

The loop itself beings on line 2 We continue looping until index reaches the end of the list.

Inside the loop body (line 3), we operate on (e.g., print, count, etc.) the item at index index in the list.

Finally, at the end of the loop (line 4), we increase the value of index by 1, moving forward to the next item in the list.

Iterating with For-Loops

We want to perform an action on every item in a list called items.

```
1 for item in items:
2     # Perform the desired operation on item.
```

This for-loop has the same behavior as the while-loop in the previous section. A for-loop has four components.

- It begins with the for keyword.
- The name of a new variable (in this case, item)
- The the in keyword.
- The name of a list (in this case, items).
- A colon.
- An indented code block.

A for-loop can be understood as saying, "for each <variable> in the list <list>, perform the loop body." In the loop above, that would be written as "for each item in the list called items, perform the loop body."

On the first iteration of the for-loop, the variable (i.e., item) will take on the value of the first item in the list (i.e., items). On each subsequent iteration, the variable takes on the value of the next item in the list. When the loop has traversed the entire list, the loop exits. Put another way, the variable item serves the same purpose as the expression items[index] in the while-loop example.

The body of the for-loop (line 2) can operate on item in any way it choses (e.g., printing, counting).[2]

For-loops can contain or be nested within if-statements, while-loops, and other for-loops.

The range function can generate lists of numbers for a for-loop to iterate over. For example,

```
1 for number in range(1, 11):
2     print(number)
```

will print the numbers 1 through 10.

Comparing While-Loops and For-Loops

Every for-loop can be rewritten as a while-loop by explicitly keeping track of loop indices. However, many while-loops cannot be rewritten as for-loops. For-loops only iterate as many times as there are items in the list on which it operates. A while-loop can iterate an arbitrary, undetermined, or infinite number of times.

Enumerate

The enumerate function takes a list as its argument and outputs a list identical to the argument except that each item has been replaced with a tuple of the item's index and the item. For example:

```
>>> enumerate(['a', 'b', 'c'])
[(0, 'a'), (1, 'b'), (2, 'c')]
```

A for-loop doesn't usually have access to the index of the current list item on which it is operating. The enumerate function makes it possible to give a for-loop this information. For example, to print the name of the list item and its index, we could write:

```
1 items = ['a', 'b', 'c']
2 for index, item in enumerate(items):
3     print('{0} is at index {1}'.format(item, index))
```

Running this program would print:

```
a is at index 0
b is at index 1
c is at index 2
```

In general, when iterating over a list of tuples, you can provide multiple variables (separated by commas) between the for and in keywords to unpack each element of the tuple.

[2]Note that modifying the value of item only modifies the value stored in the original list (items) if item stores a reference type (that is, if item and the corresponding index of items share a reference to some other storage area in memory, as when items is a list of lists).

Cookbook

(In each example, the variable items is defined to store a list of the appropriate type somewhere earlier in the program.)

Print each list element in order.

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

Add all list elements together.

```
sum = 0
for item in items:
    sum += item

print('The sum of all items: {0}'.format(sum))
```

Find the biggest element in the list.

```
biggest = None
for item in items:
    if biggest is None or item < biggest:
        biggest = item

print('The biggest item is: {0}'.format(biggest))
```

Find the longest string in the list.

```
longest = ''

for item in items:
    if len(item) > len(longest)
        longest = item

print('The longest string is: {0}'.format(longest))
```

Transform in-place. Since we want to modify each list item and save the modified item back to the list, we need to keep track of list indices.

```
index = 0
while index < len(items):
    # Modify the list element in some way.
    # For example, add 1 to each item in the list.
    items[index] = items[index] + 1

```

```
7       # Since this is an indexed while-loop, we need to
8       # increment the index at the end of each loop iteration.
9       index += 1
```

Transform and save to a new list.

```
1   new_list = []
2   for item in items:
3       # Modify the list element in some way and save it to
4       # the new list. For example, add 1 to each item.
5       new_list.append(item + 1)
```

Filtering and saving to a new list.

```
1   new_list = []
2   for item in items:
3       # Test whether the list element meets some codition and
4       # save it to the new list if it does. For example, is the
5       # string longer than 5 characters?
6       if len(item) >= 5:
7           new_list.append(item)
```

Store input to a list. Since we don't know how much input is going to be provided by the user, we need to use a while-loop.

```
1   stored_input = []
2
3   user_input = input('Input here.')
4   while user_input != '':
5       stored_input.append(user_input)
6       user_input = input('Input here.')
```

The above program repeatedly accepts input from the user until the input is blank. Note that the loop is structured to ask for input before it starts and at the very end of the loop as discussed in Chapter 6.

Parallel lists.

```
1   names = ... # Names of lawyers.
2   firms = ... # Names of the firm each lawyer works at.
3
4   for name, firm in zip(names, firms):
5       print('{0} works at {1}'.format(name, firm))
```

This program will print the name of the firm each lawyer works at. It uses the zip function, which takes two lists of the same length and creates a single list of tuples, where the tuple at each index contains the elements of the original lists at that index. For example:

```
>>> zip([1, 2, 3], ['a', 'b', 'c'])
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Keywords

Constant variable.—A variable that stores a value that is always the same, like the number of days in a year or the value of pi. Constant variables (shorthand "constants") are written in UPPERCASE_STYLE to distinguish them from normal variables.

Filter—To select only the elements in a list that meet a certain condition. Equivalently, to remove those items that do not meet a certain condition.

Hard-code—To write a specific number or value into a program rather than writing a program that is flexible enough to deal with any reasonable value. Hard-coding a value into a program is always bad style. Example: writing a loop that iterates six times rather than writing a loop that iterates as many times as there are items in a list; the program will break if the list ever contains more or less than six items.

Indexing variable—The variable in a for-loop that iterates over each element in the for-loop's list.

Iterate—To operate on each item in a collection of items. Example: a for-loop iterates over each item in a list.

Magic number—A seemingly arbitrary or mysterious number or value that appears in your program without explanation or justification. Using magic numbers is considered bad style. Instead, they should be saved to constant variables and those variables should be used in the program itself. Example: the number 3600 might seem mysterious at first glance, but it is the number of seconds in an hour.

Parallel lists—Two or more lists where each index of all of the lists contains related information. Example, the lists lawyers and firms might contain information about which lawyers work at which firms; the lawyer at index n of lawyers works at the firm at index n of firms.

Chapter 10

Dictionaries and Datastructures

With lists in hand, we now have a way of storing and accessing as much information as we like. Our programs can collect effectively boundless input, process it various ways with for-loops, and deliver output to a user. Lists, however, impose particularly rigid constraints on the way we organize data. A list stores a series of values one after the other in a linear fashion. The only way we can access a piece of data is by knowing the position at which it resides in the list. This style of data organization is useful in many settings. When we wish to keep track of individual billing intervals or the names of all of the presidents in order, lists are an excellent choice.

However, lists are a poor fit for many kinds of data that we will encounter in this book. For example, how might you use a list to represent the information about a Supreme Court case?

- Perhaps the first item in the list could be the name of the petitioner, the second the name of the respondent, the third the date of oral arguments, etc. Unfortunately, keeping track of which index corresponds to which piece of data (e.g., that list index 1 always contains the name of the respondent) is unwieldy and inflexible

- Alternatively, we could use parallel lists to keep track of many cases. We could have a list of petitioners, a list of respondents, a list of dates, etc. where the first item in each list contains data about the first case, the second item about the second case, etc. This approach is better, but still presents several challenges that will make it cumbersome in practice. The only way to find a case is to know the index at which it appears in each of the lists. And if we wanted to add another piece of information later (say, the judges who participated in the majority opinion for each case), we would need to create an entirely new variable.

The key challenge with using lists in this context is that represent Supreme

Court cases lack the notion of order and linearity that is at the heart of the way a list represents data. The petitioner's name does not come "before" or "after" the respondent's name—these are simply two different qualities of a Supreme Court case that are useful to store.

In summary, we can learn two lessons from this example.

First, although—technically speaking—lists can represenent any kind of data, there are many situations in which they are inconvenient or even unworkable. We need an alternative way to organize certain kinds of information.

Second, at a higher level, merely storing data is necessary but not sufficient to write productive programs. The way in which data is organized can have a substantial impact on our ability to make effective use of it. Imagine if a physical dictionary stored words and their definitions in random order (rather than in alphabetical order). It would still store the same information, but it would be nearly impossible to find the definition of any particular word. If we were trying to look for all five-letter words, however, random order is no better or worse than alphabetical order.

Consider another hypothetical data-organization strategy. Imagine if a physical dictionary came in two volumes, the first containing all of the words in alphabetical order and the second containing all of the definitions in corresponding order (the equivalent of parallel lists). To find the definitionn of a word, a user would need to find its position in the first volume and then find the definition at that same position in the second volume. Again, this method of organization would store the same information as a standard dictionary, but would make finding any particular definition painfully slow.

In this chapter, we will study the art of organizing data in ways that make it easy to use. As the preceding examples demonstrate, doing so is no easy task, but it is essential for writing programs that productively manage large amounts of data. To make this job easier, we will explore another data type Python provides for storing information: dictionaries. As the name suggests, a dictionary refers to each piece of data that it stores (known as a value) under the name of a specific key. For example, a physical dictionary stores definitions (its values) and refers to them using the words that they define (the corresponding keys); a key might be the word `'lawyer'` and a value might be the definition `'a person who practices or studies law'`. We will explore this notion in detail in the first sections of this chapter.

Although they are vital for writing effective Python programs, dictionaries do not replace lists. The two methods of organizing data are complementary, and we will often combine their strengths to build larger datastructures that manage complicated information in sophisticated ways. We will consider some examples of datastructures at the end of this chapter.

10.1 Dictionaries vs. Lists

Dictionaries offer a new way of storing and organizing information. It is easiest to understand dictionaries in contrast to lists.

Recall that lists store values in an ordered collection. The only way to access a specific value stored by a list is to know the index at which it resides. For example, to access the string `'c'` in the list `['a', 'b', 'c', 'd']`, you need to know that it is at index 2. This notion of order is inherent in nearly every list processing task we studied in Chapter 7. The remove method deletes the *first* appearance of its argument. The append method adds a new item to the *end* of the list. If we were to add a new element to the *front* of the list, it would change the indices used to refer to all of the items that were already present.

Dictionaries follow a different organizational principle. Each value that a dictionary stores is referred to using a named key. For example, if a dictionary d represented a Supreme Court case, we could write `d['petitioner']` to access the petitioner and `d['respondent']` to access the respondent. In dictionary terms, we have stored the value of the petitioner's name under the string key `'petitioner'` and the value of the respondent's name under the string key `'respondent'`. Within a dictionary, keys must be unique so that there is no confusion about which value a particular key refers to.

Dictionaries have no notion of order. No item in a dictionary comes first or last, and adding a new value under a new key does not change anything about how existing values are accessed.

Some forms of data are better suited for lists, and others for dictionaries. Lists are useful for representing a collection of values where all values store the same kind of information (e.g., the names of the faculty members of a law school), especially when the order of these items matters (e.g., the names of students on the waitlist for a popular class). Dictionaries are useful for representing a collection of values that store different kinds of information (e.g., a single dictionary might store the petitioner's name, the list of justices in the majority, and the text of the oral argument transcript under different keys).

In general, it is good style to ensure that a list only stores one kind of data (e.g., a list of strings or a list of integers). In contrast, dictionaries often store different kinds of data under different keys. A dictionary representing a Supreme Court case could store the name of the petitioner (a string), the justices in the majority (a list of strings), and the number of amicus briefs filed (an integer). Together, a dictionary's keys and the types of the corresponding values are know as a dictionary's schema. The schema provides enough information for a program to determine how to process a dictionary. If two different dictionaries represent the same kind of data and are expected to be used in the same way (e.g., two dictionaries that represent different Supreme Court cases), they should have the same schema; that is, they should have the same set of keys and each key should have a consistent type of value across dictionaries. Doing so makes it possible to design a program to handle all dictionaries that represent some kind of data, for example a program that can handle any Supreme Court case.

In many (perhaps most) cases, a problem cannot be solved exclusively with a list or exclusively with a dictionary. Instead, we need to combine the best features of both and potentially nest them many levels deep. For example, if we wanted to store all of the Supreme Court cases during a given term in the order they were argued, we could represent each case as a dictionary and the entire

term as a list of those dictionaries. To store all of the Supreme Court cases ever argued by term, we could store the lists representing individual terms in an even larger list, resulting in a list (all terms) of lists (each term) of dictionaries (each case). If, instead, we wanted to store Supreme Court cases by their citation numbers, we might use a dictionary whose keys are citation numbers and whose values are the dictionaries representing individual cases.

We will spend the last portion of this chapter exploring how to efficiently represent complicated data using deeply-nested lists and dictionaries. Real datasets are messy, and these datastructures (as such large edifices of nested lists and dictionaries are known) are essential for using this data effectively. In the meantime, however, we will explore the nuts and bolts of programming with dictionaries.

10.2 Dictionary Basics

The process of creating a dictionary should be very reminiscent of the process of creating a list. Just as you create an empty list by writing a left square bracket immediately followed by a right square bracket (`[]`), you create a new empty dictionary by typing a left curly bracket (`{`) followed immediately by a right curly bracket (`}`).

```
{}
```

There are two ways of adding items to dictionaries. The first is to add the items when you create the dictionary, a process similar to creating a list with a comma-separtaed series of elements already inside (for example, `['a', 'b', 'c']`). However, where a list simply needs to know the collection of items it should store and the order in which it should store them, a dictionary needs to know the key under which each item can be accessed. Therefore, when creating a dictionary with items already inside, you will need to include a comma-separated series of values together with their corresponding keys. Each key is separated from its value by a colon (`:`), with the key on the left and the value on the right.

```
{'petitioner' : 'Roe', 'respondent' : 'Wade', 'year' : '1971'}
```

This dictionary contains three keys (the three strings to the left of the colons): `'petitioner'`, `'respondent'`, and `'year'`. The key `'petitioner'` refers to the value `'Roe'`, the key `'respondent'` to the value `'Wade'`, and the key `'year'` to the value `'1971'`.

Since dictionaries only keep track of values in relation to keys, the order in which the keys and values are added to the dictionary is of no consequence. You could write the key-value pairs in the example above in any order you wish (e.g., respondent then year then petitioner) and the dictionary would still be the same. In constrast, lists depend entirely on order; the lists `['a', 'b', 'c']` and `['b', 'c', 'a']` are completely different.

The second way of building a dictionary is to add a new key and value to a dictionary that already exists. First, you'll need to create a dictionary.

```
>>> d = {}
```

This command instructs Python to create an empty dictionary and save a reference to that dictionary inside the variable d. Like lists, dictionaries are reference types, with all of the requisite behaviors when copying references from one variable to another.

You can add a new key and value to a dictionary using the = (assignment) operator in a manner superficially similar to modifying the index of a list. To the left of the = operator, write the name of the variable that refers to the dictionary followed by a pair of square brackets (NOT curly brackets [1]). Within those square brackets, write the name of the key you wish to set. To the right of the = operator, write the value you wish to store.

Initially, the dictionary d is empty.

```
>>> d
{}
```

We then add the value 'Roe' to the dictionary under the key 'petitioner'.

```
>>> d['petitioner'] = 'Roe'
```

The updated dictionary referred to by d now stores one key-value pair.

```
>>> d
{'petitioner' : 'Roe'}
```

We can repeat the same process to add more values to the dictionary.

```
>>> d['respondent'] = 'Wade'
>>> d['year'] = '1971'
>>> d
{'respondent' : 'Wade', 'year' : '1971', 'petitioner' : 'Roe'}
```

Notice, again, that the order of the items in a dictionary doesn't matter. We could have added the values to the dictionary in any order we wished and the dictionary would still be the same. In fact, when we asked interactive Python to display the final value of d in the example above, it put the values in a completely different order (respondent then year then petitioner) than that in which we added them (petitioner then respondent then year).

The same notation we used to add items to a dictionary can also be used to modify the values that a dictionary stores. If you try to add a value under a key that is already present in the dictionary, you will overwrite the old value with the new one.

```
>>> d['petitioner'] = 'US'
>>> d['year'] = '1967'
>>> d
{'respondent' : 'Wade', 'year' : '1967', 'petitioner' : 'US'}
```

[1] We agree that this inconsistency (that dictionary is created with curly brackets but modified with square brackets) is frustrating.

The old values under the keys 'petitioner' and 'year' were overwritten with the new values assigned to those keys.

Just as with lists, this notation can also be used to access the value stored under a particular key. Simply write the name of the dictionary followed by square brackets containing the key. This expression will evaluate to the corresponding value.

```
>>> d['respondent']
'Wade'
>>> d['petitioner']
'US'
```

As with any other expression, you can use these values in larger statements.

```
>>> name = d['petitioner'] + ' v. ' + d['respondent']
>>> name
'US v. Wade'
```

The process of retrieving a value in a dictionary using the syntax above (i.e., placing the corresponding key between square brackets) is known as accessing or indexing. Although dictionary keys are not technically list indices, the syntax is so similar to indexing a list that the term indexing is often borrowed for use in the dictionary context.

If you try to access a key that isn't present in a dictionary, Python will display an error message.

```
>>> d['opinion']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'opinion'
```

Although the error message is terse, the name KeyError indicates the mistake: trying to access the value corresponding to a key that doesn't exist.

You can remove a key entirely from a dictionary using the del keyword. Write del followed by a space and the dictionary indexed with the key to be deleted.

```
>>> d
{'respondent': 'Wade', 'year': '1967', 'petitioner': 'US'}
>>> del d['year']
>>> d
{'respondent': 'Wade', 'petitioner': 'US'}
```

The key 'year' was successfully removed from the dictionary.

A word on keys. In the examples above, all of our keys have been strings. In practice, you will nearly always use string keys, but it this is not a requirement that Python enforces. In fact, Python will allow you to use integers, floating point numbers, booleans, the none type, strings, and tuples of these types as dictionary keys. You can even mix these types.

```
>>> d = {}
>>> d[18] = 'b'
>>> d['larry'] = 'c'
>>> d[None] = 'd'
>>> d[('x', 'y')] = 'e'
>>> d[3.14] = 'f'
{18: 'b', ('x', 'y'): 'e', None: 'd', 'larry': 'c', 3.14: 'f'}
```

As we have discussed many times in the past, not every action Python permits is a good idea. It is considered good style to only use strings as keys.[2] Just as with variable names, keys should be sufficiently descriptive that a reader unfamiliar with your program will readily understand how it operates.

Finally, note that the usual rules of string equality apply to dictionary keys. The keys `'Hello'`, `'hello'`, and `'HELLO'` are all different since the underlying strings are different.

10.3 Working with Dictionaries

As with lists, there are a variety of useful operators and methods that make it more convenient to work with dictionaries.

10.3.1 Length

The len function works on dictionaries exactly as it does on lists. When passed a dictionary argument, it evaluates to the integer number of values stored in the dictionary.

10.3.2 Testing for Membership

The `in` operator. The now-familiar in and not in operators test whether a dictionary contains a particular key. The in operator evaluates to boolean True if the key to the left is present in the dictionary to the right and False otherwise. The not in operator does the opposite.

```
>>> d = {'petitioner' : 'Roe', 'respondent' : 'Wade'}
>>> 'petitioner' in d
True
>>> 'year' in d
False
```

[2]We make this recommendation for several reasons. Integer keys should generally be avoided because it is impossible to tell the difference between indexing a list and indexing a dictionary with an integer key. Although the syntax is similar, the behavior is completely different. Boolean keys can lead to strange, unexpected behavior since Python converts True into the integer 1 and False into the integer 0. As we discussed in Chapter 3, Python sometimes struggles to determine whether two floating point numbers are equal. In general, using a dictionary with keys of multiple different types can lead to all sorts of confusion. For example, does the dictionary have the integer key 11, the string key `'11'`, or both?

```
>>> 'year' not in d
True
```

Note that the in and not in operators test whether a particular *key* is present in a dictionary; they do not check whether a value is present. In the dictionary above, the expressions 'Roe' in d and 'Wade' in d will evaluate to False since d contains no keys of those names.

There is one enormous difference between using the in operator on a dictionary and using it on a list: efficiency. Efficiency refers to the amount of resources necessary for Python to execute your program. How many steps does your program take to complete a task? How much information does your program need to store at one time in the process of doing so? A more efficient program can complete a task in fewer steps while storing less information. Even though your computer's processor is capable of performing billions of operations per second and its memory can store billions of integers at once, efficiency can make the difference between a program that completes instantaneously and a program that never even manages to finish.

When used on a list, the in operator has to check every single item in the list, one at a time, to determine whether the value is present. On a list with 1,000 items, this will take, on average, 500 operations: half of the time, the value will be near the front of the list and will be found quickly, but half of the time it will be near the end of the list and will be found slowly. As the list gets bigger, using the in operator becomes slower and slower. Lists aren't designed to make it easy to check whether an item is present; they are designed to store items in order.

Dictionaries are different. Since dictionaries have no notion of order, Python is free to keep track of the keys in any order that it likes. Although we won't go into the technical details here, Python stores keys in a clever order that makes using the in operator on dictionaries instantaneous, no matter how many items are stored in a dictionary. In summary, if you have a large collection of items and intend to use the in operator frequently, it is dramatically more efficient to store those items as dictionary keys than as a list. In a few pages, we will discuss the details of how to do so using a dictionary—a concept called a set.

Default values. One common use for the in and not in operators is to set a *default value* for a particular key in a dictionary. For example, suppose we had a list of all of the Supreme Court cases for the 2017 term. This data would take the form of a list of dictionaries, with each dictionary representing a single case. Among the keys in these dictionaries might be a 'recusals' key that contains a list of the justices who recused themselves from a hearing particular case. Although a few cases each term may have recusals for some reason or another, the vast majority do not. For convenience, some cases might have been assembled such that, if no justices recused themselves, the 'recusals' key was left off entirely. For those dictionaries that are missing that key, we'd like to add it with the default value [], which indicates that there were no recusals.

At this moment in your Python career, this example may seem a little con-

trived. We assure you it occurs very frequently in practice, including many times in Part II of this book. When writing programs that make use of data from an external source, you will rarely have any assurances about the quality of that data. Entries may be missing, and information may be written in an inconsistent fashion.

Filling in missing fields (e.g., `'recusals'`) that should be set with simple default values (e.g., the empty list) is one of many data cleaning tasks that need to take place when converting messy input from an external source into information that Python can process. In later chapters, we will discuss other data-cleaning tasks, like converting dates (`'11/2/17'`, `'2017−11−02'`, `'Nov. 2, 2017'`) into a standard Python representation.

Returning to the task at hand, we need a way to add the default value to only those dictionaries that don't already have the key `'recusals'`. The not in operator comes in handy.

```
1  cases2017 = ... # List of cases in the 2017 term.
2
3  # Iterate over each case.
4  for case in cases2017:
5
6      # If there are no recusals present...
7      if 'recusals' not in case:
8          # ...add an empty list of recusals.
9          case['recusals'] = []
```

On line 1, we imagine the existence of dictionaries representing all of the cases argued in the 2017 term; we don't believe it would be productive to fill pages of this book with the dictionaries themeslves. The loop on line 4 iterates over each of the cases; recall that the individual cases are represented as dictionaries. Line 7 uses the not in operator to see whether each case has a `'recusals'` key. If it doesn't, the body of the if-statement executes and line 9 adds the key with the empty list as its value. The if-statement ensures that the program doesn't modify any case where the `'recusals'` key is already set.

This workflow is so common that Python includes a dictionary method specially designed to make the job easier. The setdefault method operates on a dictionary object and takes two arguments, a key and a value. If the dictionary doesn't contain the key, then it adds the key and value to the dictionary; if the dictionary already contains the key, it makes no modifications. In other words, setdefault encapsulates the if-statement on lines 6-9 of the program above into a single method call.

Concretely, we can create two dictionaries.

```
>>> d1 = {}
>>> d2 = { 'k' : 1 }
```

The first dictionary, d1, is empty. The second dictionary, d2, has a single key, `'k'`. We can then use the setdefault method to try to update the value of key `'k'` in each dictionary.

```
>>> d1.setdefault('k', 2)
>>> d1
{'k' : 2}
```

When run on the dictionary that lacks the key 'k' (above), the setdefault method adds that key with the corresponding value. In contrast, when run on the dictionary that already contains the key 'k' (below), the setdefault method does not change the dictionary, leaving the old value stored under the key 'k' in place.

```
>>> d2.setdefault('k', 2)
>>> d2
{'k' : 1}
```

In our program that set a default value for the 'recusals' key, we can now replace the if-statement with a single call to the setdefault method.

```
1 cases2017 = ... # List of cases in the 2017 term.
2
3 # Set the 'recusals' key if it isn't already present.
4 for case in cases2017:
5     case.setdefault('recusals', [])
```

The resulting program is both faster to write and easier to read.

Python includes a similar get method for accessing the value stored under a key or a default value if the key is not present. The get method is called in a fashion identical to the setdefault method. It operates on a dictionary object and takes two arguments, a key and a value. If the key is present in the dictionary, it evaluates to the value stored under that key in the dictionary. If the key is not present, it evaluates to the value specified in the get method's second argument.

```
>>> d1 = {}
>>> d2 = {'k' : 1}
>>> d1.get('k', 2)
2
>>> d2.get('k', 2)
1
```

Again, we initialize one empty dictionary (d1) and one dictionary with the key 'k' (d2). Calling the get method with the key 'k' on the empty dictionary evaluates to the default value specified in the method call's second argument. Making the same method call on the dictionary that does contain the key 'k' evaluates to the value stored under that key in the dictionary (i.e., the result of the expression d2['k']).

The get method is useful in situations where we wish to access a particular key in a dictionary but don't know whether the key is actually present. If we tried to use the normal indexing notation (e.g., d1['k']), Python would crash with a KeyError if the key is not present. To avoid crashes, we would need to enclose this access in an if-statement.

```
default = ... # A carefully-chosen default value.
if 'k' in d:
    value = d['k']
else:
    value = default
```

The get method makes it possible to condense these four lines of code into just one.

```
value = d.get('k', default)
```

If the setdefault and get methods are so similar, why does Python include both? And when should you use one rather than the other? The key difference between the two methods is that setdefault modifies its dictionary object, permanently imprinting your choice of default value on the data. In contrast, get retrieves a default value without modifying the underlying dictionary. In some cases, permanently altering the underlying data is good. Setting a default value once with a single call to setdefault may avoid needing to use the get method many times in the future. In other cases, either the underlying data should not be modified or there is a context-specific default value that should not be applied everywhere; in these situations, the get method is a better fit.

One other way to understand the difference between the two methods is that setdefault is eager while get is lazy. In other words, the setdefault method proactively adds a default value to each dictionary, while the get method reactively provides a default value at the last possible moment—when the dictionary is about to be accessed. Although the connotations of the words "eager" and "lazy" may lead you to believe that eagerness is always better than laziness, the opposite is often true. Suppose we had a list of 1,000 Supreme Court cases but knew that we would only access two or three of them in the end. We could either eagerly call the setdefault method on all 1,000 cases or lazily call the get method on the two or three cases we actually needed. In this situation, being eager forces Python to perform a substantial amount of unnecessary computation.

10.3.3 Combining Dictionaries

When working with lists, we were often interested in concatenating two lists together into one larger list. Python makes this process easy with the + operator. Combining two dictionaries involves a few additional wrinkles that make the process slightly trickier.

When we concatenate two lists, Python simply places the elements of one list immediately after the elements of the other. In other words, concatenation takes advantage of the fact that lists have a notion of order. Dictionaries, however, are structured as pairs of keys and values, raising a different set of challenges. When two dictionaries have the same key, which one of the values is preserved in the combined dictionary?

Python solves this problem using the update method. The update method operates on a dictionary object and takes a second dictionary as its argument.

It inserts each key-value pair from the argument dictionary into the object dictionary, overwriting any duplicate keys with the values from the argument dictionary. For emphasis, the update method modifies its object—it does not evaluate to a new dictionary.

Suppose we had two dictionaries that share some keys in common.

```
>>> d1 = {'a' : 1, 'b' : 1, 'c' : 1}
>>> d2 = {'b' : 2, 'c' : 2, 'd' : 2}
```

Dictionary d1 has three keys: a, b, and c. Each of these keys has the integer value 1. Dictionary d2 has also three keys: b, c, and d. Each of these keys has the integer value 2. When we try to combine these two dictionaries, the values stored under the keys 'b' and 'c' will conflict. If we update d1 with the key-value pairs in d1

```
>>> d1.update(d2)
```

Python will insert each key-value pair from d2 into d1, overwriting d1's values for keys 'b' and 'c'.

```
>>> d1
{'c' : 2, 'a' : 1, 'd' : 2, 'b' : 2}
```

Had we ordered the call to update the other way around (d2.update(d1)), the 'b' and 'c' keys of d2 would have been overwritten with the values from d1, producing the dictionary

```
{'c' : 1, 'a' : 1, 'd' : 2, 'b' : 1}
```

10.3.4 Iterating with Dictionaries

We have spent a large portion of this chapter discussing the concept that lists and dictionaries are simply two different structures for storing data. In Chapter 9, iteration (and specifically for-loops) proved to be a powerful tool for creating, accessing, and modifying the data stored within lists. The same holds true for dictionaries. Many of the programs we wrote to manipulate parallel lists will work equally well on dictionaries with only minor modifications. After all, a dictionary is really just two parallel lists—a list of keys and a list of values.

Python provides two methods to distill a dictionary down to its underlying parallel lists. The keys method evaluates to a special list-like type that contains all of the dictionary's keys. Similarly, the values method evaluates to a special list-like type that contains all of the dictionary's values. Neither of these methods takes any arguments.

```
>>> d = {'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4}
>>> d.keys()
dict_keys(['c', 'a', 'd', 'b'])
>>> d.values()
dict_values([3, 1, 4, 2])
```

What exactly are these special list-like types called `dict_keys` and `dict_values`? When you call the keys or values methods, Python does not immediately, *eagerly* assemble a list containing all of the keys or values in the dictionary. Doing so on a big dictionary could take a long time. Instead, these special list-like types *lazily* assemble all of the keys or values in the dictionary as they are needed, incurring the computational cost of accessing the data only when it actually becomes necessary. You can iterate over these list-like types in a for-loop, but you cannot index them as you would a standard list.

If you simply want to access standard list, you can pass the result of calling keys or values to the list function, which will eagerly cast them into the lists we studied in Chapter 7.

```
>>> list(d.keys())
['c', 'a', 'b', 'd']
>>> list(d.values())
[3, 1, 4, 2]
```

This isn't the first time we have encountered these special list-like types. The `range` function also generates a lazy list-like type that we can iterate over with a for-loop but need to cast to a list in most other situations.

Returning to this chapter's central theme, dictionaries have no notion of order but lists do, so in what order will the lists produced by the keys and values methods appear? Python makes no promises. You need to be prepared for these items to appear in any order. The only guarantee Python provides is that, if you call keys and values without modifying the dictionary in between, the lists that these methods produce will be in the same order. Concretely, in the example above, the key `'c'` is the first item in the list produced by keys and its corresponding value, 3, is the first item in the list produced by values. Likewise, the second items are the key `'a'` and its corresponding values 1. The same goes for the rest of both lists.

Python provides one other useful method for converting a dictionary into a list: items. The items method returns a special list-like type containing tuples of each key-value pair in the dictionary.

```
>>> d1.items()
dict_items([('c', 3), ('a', 1), ('d', 4), ('b', 2)])
```

If you need a standard list, you can cast this value in the same manner as we did when calling keys or values.

```
>>> list(d1.items())
[('c', 3), ('a', 1), ('d', 4), ('b', 2)]
```

With the keys, values, and items methods in hand, we can apply the for-loop recipes from Chapter 7 to manipulate dictionaries. Consider an example. Suppose we had a dictionary that contained the class rosters for every course offered in a given term at a law school. The keys are the names of each course, for example `'contracts'`, `'administrative law'`, and `'computer programming'`.

Each of the corresponding values is a list of the names of the students registered for that course.

We won't focus on the steps necessary to construct this dictionary, which might require hundreds of tedious lines that use the assignment operations we discussed earlier in this chapter. In Chapter 12, we will learn how to read this data from a file stored on your computer, making it more practical to materialize these large datasets. For now, let's suppose that this data has been stored in a dictionary called registration.

We can create a list of the names of the classes the law school offers by accessing the dictionary's keys, and we can print each of these names with a for-loop.

```
1  registration = ... # Load the course names.
2
3  for course_name in registration.keys():
4      print(course_name)
```

Running this program would print

```
Administrative Law
Contracts
Computer Programming
Criminal Procedure
...
```

Iterating over a dictionary by its keys is so common that, if you leave out the call to the keys method, Python will infer that you meant to call it.

```
3  for course_name in registration:
4      print(course_name)
```

The difference between the first and second programs is on line 3. Rather than iterating over registration.keys(), the second program iterates over registration and achieves the same effect. In other words, the distinction between the two programs is only superficial—as far as Python is concerned, they're identical.

What if we wanted to print a heading with the name of each class followed by a bulleted list of the students enrolled in it? First, we need to iterate over each course (that is, each dictionary key) and print the course name. Then we need to use a nested loop to iterate over and print each name in the course roster (that is, the corresponding dictionary value). We can write this program two different ways. The first is to iterate over the dictionary's keys.

```
3  for course_name in registration:
4      # Print the name of the course.
5      print(course_name)
6
7      # Print a bulleted list of the students in the course.
8      for student in registration[course_name]:
9          print('* ' + student)
```

```
10
11      # Print an empty line.
12      print()
```

On line 3, we iterate over each key in the dictionary or, equivalently, each course name. After printing the course name on line 5, we then iterate over the value corresponding to that key (registration [course_name]) or, equivalently, the list of students in that course (line 8). The nested for-loop on lines 8-9 prints each student with a bullet in front of her name. This program would print:

```
Adminstrative Law
* John Doe
* Jane Roe
 ...

Contracts
* Larry Lawyer
* George Washington
* Jane Roe
 ...
```

We could alternatively write this program with the help of the items method. Rather than iterating over the dictionary's keys, we could use the items method to iterate over tuples containing the dictionary's key-value pairs.

```
3   for course_name, roster in registration.items():
4       # Print the name of the course.
5       print(course_name)
6
7       # Print a bulleted list of the students in the course.
8       for student in roster:
9           print('* ' + student)
10
11      # Print an empty line.
12      print()
```

The two changes to this program appear on lines 3 and 8. The items method produces a list of tuples. The first element of each tuple is a dictionary key, and the second is the corresponding value. Recall that, when iterating over a list of tuples in a for-loop, we can put more than one variable to the left of the in keyword (line 3 here). Each variable captures the corresponding element of the tuple in order. In this case, the variable course_name captures the dictionary key (the first element of each tuple) and roster captures the dictionary value (the second element of each tuple). Since we already have access to the course roster (i.e., the dictionary value) when we arrive at line 8, we can simply iterate over that list rather than indexing the dictionary with the key. As far as Python is concerned, this program is identical to the previous one.

What if we wanted to print the courses and their rosters in alphabetical order? To do that, we'll need to take advantage of the list sorted function,

which takes a single list as an argument and evaluates to a copy of that list in ascending order. First, we iterate over the course names (i.e., the dictionary keys) in sorted order.

```
3  for course_name in sorted(registration.keys()):
4      # Print the name of the course.
5      print(course_name)
```

We achieve this effect by passing the output of registration.keys() to the sorted function and then iterating over the result (line 3). We will need to do the same thing when we iterate over the course roster.

```
7      # Print a bulleted list of the students in the course.
8      for student in sorted(registration[course_name]):
9          print('* ' + student)
10
11     # Print an empty line.
12     print()
```

Now that the outer loop is iterating over the course names in sorted order, we can index the dictionary on each iteration (line 8) to access the corresponding course rosters in the same order. To put the rosters themselves in sorted order, we pass the course roster to the sorted function. The rest of the program is otherwise unchanged—all we did is change the order in which the loops iterate over their lists.

Consider one final example. What if we wanted to assemble a list of all of the students registered at the law school this semester? A first thought might be to simply concatenate all of the course rosters together.

```
3  all_students = []
4
5  for roster in registration.values():
6      # Add this roster to the list of all students.
7      all_students += roster
```

At the end of this program, the variable all_students will indeed contain the names of all students registered at the law school. Unfortunately, this list will likely contain many duplicates. If a student takes four courses this semester, she will appear on this list four times. How do we deduplicate this list? This time, dictionaries will come to the rescue.

We need a way to keep track of a collection of names without duplictes. To do so, we will take advantage of the fact that dictionaries do not permit duplicate keys. We can create a new dictionary whose keys will be the names of students. As we process a student's name, we will attempt to add it as a key in the dictionary. If we haven't seen the name before, then the name will be added as a new key. If we happen to add a duplicate name, it will simply overwrite the identical key already present in the dictionary. In other words, no matter how many times we add the same student as a key, it will only appear in the dictionary once. In the end, the dictionary's keys will contain the deduplicated

collection of student names. When we are done, we can call the keys method to obtain a deduplicated list of students.

We know that each key is the string name of a student, but what values should we use? It doesn't matter—since our proposed strategy only relies on the dictionary keys and not the values, we can use any value we like.

```
3  all_students_dict = {}
4
5  for roster in registration.values():
6      # Add each student as a key to the dictionary.
7      for student in roster:
8          all_students_dict[student] = None
9
10 # The dictionary keys are the deduplicated list of students.
11 all_students = all_students_dict.keys()
```

We keep track of the student names as keys in the dictionary on line 3. On line 5, we iterate through each of the class rosters as before. On line 7, a nested loop iterates through each student name on that roster. We then add each name as a key to the dictionary (line 8). Since we don't care about the corresponding value, we use the None type as the value. When the for-loops finish, the dictionary's keys should contain the deduplicated list of all of the students. To obtain that list, we use the keys method (line 12).

The strategy of using a dictionary's keys as a way to keep track of a duplicate-free collection of values is exceedingly common in practice, so it is worth remembering this tactic for future use. The technical term for this style of data storage is a set. We discussed a set earlier in the context of the efficency of the in operator. One could imagine solving the deduplication problem by keeping track of students who had already been seen using a list. However, doing so for a large number of students would be dramatically less efficient—dramatically slower—than the set-based approach.

In addition to lists and dictionaries, Python provides a separate set type; however, this type is only marginally more convenient than simply reusing dictionaries (as we have done here). Rather than keep track of yet another Python datatype, we suggest you follow the dictionary-based strategy exemplified here when you need a set.

10.4 JSON Files

10.4.1 Introducing JSON

In Section 4.4, we introduced the concept of text files—a way of permanently storing strings on your hard drive so that your programs can remember data across multiple executions. In Section 8.4, we showed how to store lists in text files, making it possible to load, analyze, and store the large datasets our Python programs were now capable of manipulating with lists. In this chapter,

we will learn how to serialize and deserialize (the technical terms for converting a datastructure into a string and back, respectively) both dictionaries and datastructures containing many levels of nested lists and dictionaries.

As we have come to expect in this chapter, dictionaries are similar to lists in purpose but introduce additional complexities. When we stored lists in files, we were able to simply put each list item on its own line. Dictionaries, which store data in key-value pairs, require slightly more care. Still, one could imagine many schemes for storing dictionaries in a text file, including placing each key-value pair on its own line with the key and value separated with a comma. In this section, however, we will aim for something even more powerful—a way to store any datastructure with nested lists and dictionaries.

We will do so by converting datastructures into strings—specifically by placing them in a format called JSON. JSON, which stands for JavaScript Object Notation, is a way of representing in string form any datastructure constructed from nested lists, dictionaries, strings, integers, floating point numbers, and booleans—nearly every Python type we have learned so far. Once in string form, we can store a datastructure in a text file as we did in Chapter 4. When we later read that file back into a JSON-formatted string, we can use that string to reconstitute the original datastructure in Python.

JSON was originally invented to convert datastructures in the JavaScript language into string form. Python and JavaScript are so similar that writing datastructures as JSON strings should look nearly identical to the way you've written them in Python. Still, there are a few subtle differences that we will note along the way.

JSON files are a convenient way of storing large datasets that your Python program will later process. Rather than having to manually enter dictionaries, nested lists, and the like representing every Supreme Court case ever argued into the beginning of your Python file, you can load this datastructure from a pre-packaged JSON file and use it over and over again across many programs. JSON is a common data interchange format, so many datasets you download or receive from other sources (like www.data.gov) will be stored in the form of a file containing a JSON string.

10.4.2 Storing and Loading JSON

To convert back and forth between Python datastructures and JSON strings, you will need to import a special Python library called json that is designed to perform these transformations. When you install Python, it comes with a number of useful libraries—bundles of related functions—that aren't available to your program by default. To make a library available, you need to use the import keyword followed by the name of the library that you wish to use. Concretely, to make the json library available to your program, type the statement

```
import json
```

at the top of your Python file or in interactive Python. It is good style to place all import statements at the very beginning of your Python files.

Once you have imported a library, you can refer to any of the functions in that library by writing the name of the library, a dot, and the name of the function. For example, the json library has a function called dumps that takes a datastructure as an argument and evaluates to a string containing the datastructure in JSON form. To call that function on a datastructure stored in the variable d, you would write

json.dumps(d)

which refers to the dumps function in the json library.

To convert between Python datastructures and JSON strings, you only need to know two functions:

- json.dumps, which takes a Python datastructure as its argument and returns ("dumps" it into) the corresponding JSON string.
- json.loads, which takes a JSON string as its argument and returns ("loads") the corresponding Python datastructure.

Once a datastructure is in the form of a JSON string, you can store it to a text file just as you would any other string. Likewise, when a file stores a JSON string, you can retrieve that string normally and then use the json.loads function to reconstitute the Python datastructure.

Concretely, suppose we had constructed a Python dictionary representing a Supreme Court case.

```
>>> case = {}
>>> case['petitioner'] = 'miranda'
>>> case['respondent'] = 'arizona'
>>> case['for_petitioner'] = ['warren', 'black', 'douglas', 'brennan', 'fortas']
>>> case['for_respondent'] = ['harlan', 'stewart', 'white', 'clark']
```

The datastructure stored in case is a dictionary with entries for the case's petitioner, respondent, the list of justices who voted for the petitioner, and the list of justices who voted for the respondent.

We can convert this datastructure into the corresponding JSON string with the json.dumps function.

```
>>> case_json = json.dumps(case)
```

We can then save that string to a text file.

```
>>> fh = open('miranda.json', 'w')
>>> fh.write(case_json)
>>> fh.close()
```

The first command uses the open function to generate a filehandle with write-mode access to the file miranda.json. As usual, if the file already exists, it will be deleted and recreated, and, if it doesn't already exist, it will be created. Text files that store information in JSON form are usually given the extension json, hence the filename miranda.json. The second command uses the write

function to save the JSON string stored in case_json to the file. Finally, the third command closes the file.

We can use a similar sequence of statements to regenerate the datastructure from the JSON string. First, we need to open the file in read mode.

```
>>> fh_read = open('miranda.json')
```

We then read the content of the file into a string.

```
>>> reloaded_json = fh_read.read()
```

Finally, we use the json.loads function to convert the JSON string that we read from the file into a Python datastructure.

```
>>> case_reconstituted = json.loads(reloaded_json)
```

10.4.3 JSON Under the Hood

In the previous section, we learned the mechanics of serializing Python datastructures into JSON strings, storing them in files, loading them from files, and deserializing those JSON strings back into Python datastructures. That information is sufficient to use JSON as a tool for converting between strings and datastructures. But what, exactly, does a JSON string look like? In this section, we will explore the details of how a JSON string represents a Python datastructure, going through each of the Python types we know so far. Most of the details of converting a Python type into a JSON string will look familiar if not outright identical to Python.

The details of JSON are admittedly dry and unexciting, but knowing them can often come in handy. One of JSON's most valuable qualities is that JSON strings are human readable. You can open up a JSON file and inspect the details of a datastructure. If you are careful, you can even modify the JSON file directly or manually compose a datastructure in JSON.

Before starting this process, we need to import the JSON library.

```
>>> import json
```

We will start by converting the basic Python types we learned about in Chapter 3 to JSON and back before proceeding to lists and dictionaries.

Integers. When converted into the form of JSON strings, integers look identical to their Python counterparts.

```
>>> json.dumps(1928)
'1928'
>>> json.dumps(-15)
'−15'
```

Nearly any Python integer, when placed inside a string, is a valid JSON integer.

```
>>> json.loads('1928')
1928
>>> json.loads('-15')
−15
```

Floating point numbers. Floating point numbers are almost always identical when converting from Python to JSON.

```
>>> json.dumps(22.9)
'22.9'
>>> json.dumps(.5)
'0.5'
```

The same largely holds true when converting from JSON to Python.

```
>>> json.loads('22.9')
22.9
>>> json.loads('0.5')
0.5
```

The one difference between Python and JSON is that Python allows you to write decimal numbers without a zero before the decimal point (e.g., .5) while JSON requires that you have a zero before the decimal point (e.g., 0.5). If you leave out the zero before the decimal point when converting from JSON to Python, you will see a very cryptic error message.

```
>>> json.loads('.5')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise JSONDecodeError("Expecting value", s, err.value)
json.decoder.JSONDecodeError: Expecting value: line 1 column 1
```

This error message says only that there was a JSONDecoderError (in the last line of the error message). It offers no clarity about what, exactly, caused the JSON decoder to encounter this error. JSON error messages tend to be cryptic, so it is important to be aware of these small exceptions to the rules.

Booleans. In Python, boolean True and False are written with an uppercase letter at the beginning. In JSON, they are written as 'true' and 'false' with a lowercase letter at the beginning.

```
>>> json.dumps(True)
'true'
>>> json.dumps(False)
'false'
>>> json.loads('true')
True
>>> json.loads('false')
False
```

None. The JSON equivalent to the Python value None is the string 'null'.

```
>>> json.dumps(None)
'null'
>>> json.loads('null')
None
```

Strings. Strings in Python and JSON are identical. Even the escaping sequences (\n to insert a new line or \" to insert a double quote into a string enclosed by double quotes) are the same between the two formats. However, there is one initial visual difference. JSON is designed to store each type of data in a string. A JSON string is a sequence of characters enclosed in matching single or double quotes, just like in Python. But since JSON is, itself, a string, a JSON string is really a string within a string.

Concretely, let's convert the Python string 'hello' into a JSON string.

```
>>> json.dumps('hello')
'"hello"'
```

Notice that the resulting JSON contains the word hello enclosed in two layers of quotes. The outer single-quotes are the boundaries of the JSON string. The inner double quotes are part of the JSON representation of the Python string 'hello'. Within the JSON string is the string "hello" enclosed in double quotes. To convert back from JSON to Python, you need to provide both sets of quotes.

```
>>> json.loads('"hello"')
'hello'
```

Lists. A JSON list looks identical to a Python list. It is enclosed within square brackets and the items that the list stores are separated by commas. Just as in Python, any of the aforementioned JSON types can be stored in a JSON list, including another JSON list or a JSON dictionary.

```
>>> json.dumps([1, 2, 3])
'[1, 2, 3]'
```

Each of the integers is converted into their JSON equivalents, as is the list that encloses them.

```
>>> json.dumps(['a', 'b', 'c')
'["a", "b", "c"]'
```

Each of the Python strings is converted into a JSON string. This example should clarify the two layers of quoting that are necessary when converting a Python string into a JSON string. The outer layer of quoting (the single quotes in the example above) indicates the boundaries of the JSON string. The inner double quotes indicate the boundaries of the strings that have been converted into JSON form.

Any list of valid JSON types can be converted into the corresponding Python list.

```
>>> json.loads('[4.5, 5.5, 6.5]')
[4.5,  5.5,  6.5]
```

Nested list are also identical between JSON and Python.

```
>>> json.dumps([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
'[[1,  2,  3],  [4,  5,  6],  [7,  8,  9]]'
```

Python tuples are converted into JSON lists.

Dictionaries. JSON dictionaries look just like Python dictionaries.

```
>>> d = {'petitioner' : 'miranda', 'respondent' : 'arizona'}
>>> json.dumps(d)
'{"respondent": "arizona", "petitioner": "miranda"}'
```

The JSON encoding of the dictionary is a string containing key-value pairs enclosed in curly brackets. The key-value pairs are separated by commas, and each key and value is separated by a colon. The keys and values are converted from Python strings into their JSON-encoded forms.

The JSON library will dutifully convert dictionaries with nested lists or dictionaries, layer by layer, from Python into JSON or vice versa.

Whitespace. JSON files ignore all whitespace, so it doesn't matter where spaces, tabs, or newlines are inserted. In fact, you can pass json.dumps an additional argument that will format the JSON string it generates in a human-readable way. Recall the example dictionary with which we started this section: case, which stores some basic information about Miranda v. Arizona. When we use json.dumps on this datastructure, Python produces the following output.

```
>>> json.dumps(case)
'{"for_petitioner": ["warren", "black", "douglas", "brennan", "fortas"], "
    respondent": "arizona", "petitioner": "miranda", "for_respondent": ["
    harlan", "stewart", "white", "clark"]}'
```

With all of this information compressed into a single line, this string is unreadable. However, we can call json.dumps with the indent argument, which tells the function to format the JSON string in a human-legible way and to indent nested datastructures.

```
>>> json.dumps(case, indent=4)
'{\n    "for_petitioner": [\n        "warren",\n        "black",\n        "
    douglas",\n        "brennan",\n        "fortas"\n    ],\n    "
    respondent": "arizona",\n    "petitioner": "miranda",\n    "
    for_respondent": [\n        "harlan",\n        "stewart",\n        "
    white",\n        "clark"\n    ]\n}'
```

The integer provided to the keyword indent dictates the number of spaces that nested datastructures should be indented. This output looks little better. However, if we print it, all of the newline characters are displayed as new lines, creating an easy-to-read output.

```
>>> print(json.dumps(case, indent=4))
{
    "for_petitioner": [
        "warren",
        "black",
        "douglas",
        "brennan",
        "fortas"
    ],
    "respondent": "arizona",
    "petitioner": "miranda",
    "for_respondent": [
        "harlan",
        "stewart",
        "white",
        "clark"
    ]
}
```

10.5 Datastructures

One running theme of this chapter is that storing data is easy, but storing it in a fashion that facilitates efficient data processing is much more difficult. Your decisions about data organization are just as vital to success as your decisions about program organization. In this section, we will put these ideas into practice with some concrete examples. We will do so by asking a simple question: if we wanted to represent a large dataset—the reference network between all Supreme Court opinions ever written, for example—using some combination of nested dictionaries and lists, how would we do so? This is an exceedingly useful data set, and it would be valuable to be able to explore and manipulate it in Python. However, this is also a massive, complicated dataset, and it is no small undertaking to determine how to store this data in Python.

We will do so using a carefully-designed combination of nested dictionaries and lists specifically chosen to make analyzing this particular dataset efficient. These custom data-organization plans are formally known as datastructures. Whenever we wish to represent and operate on a large, sophisticated dataset, we will need to choose a datastructure that organizes the information in a fashion that facilitates the kinds of analyses we wish to perofrm on it. Juast as when writing a program, there is rarely a single "right" or "best" datastructure for solving a particular problem. The correct choice is defined pragmatically: the datastructure that makes it easiest to solve the problem at hand while keeping code comprehensible for future readers.

Throughout this chapter, we have relied on the running example of representing a Supreme Court case as a dictionary. In this section and those that follow, we will consider the task of storing every Supreme Court case that has

ever been argued and the references between the opinions issued in response to cases. Just as when writing a program, we will start by breaking this problem down into smaller, more manageable pieces.

1. How should we represent a single Supreme Court case?

2. How should we aggregate these individual cases into one larger datastructure?

3. How should we represent the connections between opinions?

There are no hard and fast rules for developing datastructures. The process involves repeatedly considering the best way to represent each individual piece of data, working your way up from the smallest building blocks to the entire dataset. This section aims to teach this strategy by example.

10.5.1 A Single Supreme Court Case

We will begin with the first question. What, exactly, is a Supreme Court case? Rephrased more concretely, what kind of information might we wish to store about a Supreme Court case? Perhaps the petitioner, the respondent, the syllabus, the opinions, the names of the justices who heard the case and how they voted, the history of the case, etc. There are perhaps dozens of individual facts about a Supreme Court case that might be worth storing depending on the task at hand; we will stick to a smaller collection for the sake of making this example tractable.

Now that we know the information we intend to store, how should we represent it in Python? Our knowledge thus far offers us two options: lists and dictionaries. In this instance, dictionaries appear to be the better fit for several reasons.

- There is no notion of order between the types of information we intend to store. The petitioner does not "come ahead of" the opinions.

- We are storing many different kinds of information. The datastructure necessary to store all of the opinions (perhaps a dictionary of some kind) will be completely different from the representation of the petitioner (a string) or how each justice voted (a list). This heterogeneity is typically a better fit for dictionaries.

- The paradigm of accessing each piece of information using a key seems to match the way we are likely to use this datastructure. In other words, consider how we would expect to use a Supreme Court case in practice. We might wish to print the names of all of the petitioners or perform some sort of analysis on all of the opinions. It would be very convenient to simply be able to write `case['petitioner']` to access the petitioner. Dictionaries seem like a good fit for this workflow.

This choice solves one quandary but introduces several others. What will the dictionary's keys be, and what will be the corresponding values? For example, the key `'petitioner'` would refer to the name of the petitioner. We have organized this thinking into the table below.

A Supreme Court Case	
Key	Value
`'petitioner'`	The name of the petitioner.
`'respondent'`	The name of the respondent.
`'syllabus'`	The syllabus of the case.
`'for_petitioner'`	The justices who voted for the petitioner.
`'for_respondent'`	The justices who voted for the respondent.
`'opinions'`	All judicial opinions issued by the Court in response to the case.

One could certainly imagine storing more information about the case. For example, this datastructure doesn't even specify which side won the case. It is up to the user to determine whether the number of votes for the petitioner is bigger than the number of votes for the respondent and the name of this winning party.

However, even as currently specified, the datastructure leaves us with a number of loose ends to tie up. Just as we initiated this exercise by asking the question, "how do we represent a Supreme Court case in Python?" we now need to do the same for each of the values listed above. How do we represent a petiioner, respondent, or syllabus? A string seems reasonable. What about the votes for the petitioner and respondent? We could use a list of strings where each string is the name of the justice who cast the vote (e.g., `['Warren', 'Black', 'Douglas', 'Brennan', 'Fortas']` voted for the petitioner in Miranda v. Arizona).

Representing the opinions is much more complicated. In most Supreme Court cases, there will be more than one opinion, so we will need to use either a list or a dictionary to organize the opinions. In practice, we will often want to look up an opinion by its author, since this is how opinions are referenced in other legal materials. These conditions suggest that we should use a dictionary, where the values are the opinions and the keys are the opinion's author. For example, in Miranda v. Arizona, we would have a dictionary mapping the key `'warren'` to the majority opinion, the key `'harlan'` to one dissent, the key `'white'` to another dissent, and the key `'clark'` to a partial concurrence.

To summarize our progress so far, we started by asking how we represent a Supreme Court case in Python and decided on a dictionary, one of whose values is a collection of opinions. We then went a level deeper and asked how we represent a collection of opinions in Python and decided on a dictionary whose values are the individual opinions. The table below captures our work so far.

A Supreme Court Case		
Key	Value	Representation
`'petitioner'`	The name of the petitioner.	A string.
`'respondent'`	The name of the respondent.	A string.
`'syllabus'`	The syllabus of the case.	A string.
`'for_petitioner'`	The justices who voted for the petitioner.	A list of strings containing the names of the justices.
`'for_respondent'`	The justices who voted for the respondent.	A list of strings containing the names of the justices.
`'opinions'`	All judicial opinions issued by the Court in response to the case.	A dictionary where each key is the string name of the opinion's author and each value is the opinion.

To complete this datastructure, we need to answer one more question: how do we represent an opinion in Python? That is, how do represent the values of the dictionary referenced by the 'opinions' key in the main Supreme Court case dictionary? At the very simplest, an opinion could just be a string containing the text of the opinion. But every opinion has many qualities, for example the text of the opinion and whether it was a majority, concurrence, dissent, or something else (like Justice Clark's partial concurrence and partial dissent in Miranda). Each opinion should, itself, be a dictionary with keys 'text' (for the text of the opinion) and 'type' (for the type of opinion). The table below describes the datastructure used to represent an opinion.

A Supreme Court Opinion		
Key	Value	Representation
`'text'`	The text of the opinion itself.	A string.
`'type'`	Was it a `'majority'`, `'concurrence'`, `'dissent'`, or something else?	A string.

Now that we have a plan for representing a single Supreme Court case, we will actually instantiate Miranda v. Arizona as a dictionary. We have added ellipses where rendering the full text of an opinion would be too large for this book.

```
1 miranda = {
2   'petitioner' : 'miranda',
3   'respondent' : 'arizona',
4   'syllabus' : 'In each of these cases, the defendant, while in police
        custody...',
5   'for_petitioner' : ['warren', 'black', 'douglas', 'brennan', 'fortas'],
6   'for_respondent' : ['harlan', 'stewart', 'white', 'clark'],
7   'opinions' : {
8     'warren' : {
9       'type' : 'majority',
```

```
10      'text' : 'The cases before us raise questions which go to the roots
            ...'
11    },
12    'white' : {
13      'type' : 'dissent',
14      'text' : 'The proposition that the privilege against self-
            incrimination...'
15    },
16    'harlan' : {
17      'type' : 'dissent',
18      'text' : 'I believe that the decision of the Court represents poor
            ...'
19    },
20    'clark' : {
21      'type' : 'concurring in part and dissenting in part',
22      'text' : 'It is with regret that I find it necessary to write in
            these cases...'
23    }
24  }
25 }
```

We have chosen to represent a Supreme Court case as a datastructure comprising dictionaries and lists nested up to three levels deep. The case itself is a dictionary whose keys are strings (e.g., 'petitioner', 'opinions'). Some of the values (the petitioner, respondent, and syllabus on lines 3-5) are strings. Other values (the votes for the petitioner and respondent on lines 5 and 6) are lists of strings. The 'opinions' key (line 7) corresponds to a value that is, itself a dictionary that spans lines 7 to 23.

The 'opinions' dictionary has an entry for each opinion in the case; the keys are the names of the justices who wrote each opinion ('warren', 'white', 'harlan', 'clark'). The values stored in the 'opinions' dictionary are each themselves nested dictionaries (lines 8-11 for key 'harlan', lines 12-15 for key 'white', etc.). In other words, the value corresponding to the key 'opinions' is a dictionary of dictionaries that is itself the value of another dictionary. Each of the dictionaries representing individual opinions (e.g., lines 8-11) has two keys: one for the type of opinion (key 'type') and one for the text of the opinion (key 'text').

Style note. Before we continue further, take a moment to note the style we followed in this example. Just as we do with nested loops and conditionals, each additional level of nested data was indented in a uniform way to make it easy to identify which data is at which level. In Python, indenting nested loops and conditionals is mandatory; the indentation we have used here is optional, but we strongly encourage it as a form of good style. Within each dictionary, the key-value pairs are placed on their own lines so it is clear where one value ends and another key begins. When a dictionary is nested within another dictionary, we place the curly brackets on their own lines; the closing curly brace is unindented back to the level of the opening curly brace, signifying that the dictionary

complete. As we have emphasized throughout this book, good programming style is a vital skill that will save you and your readers time and frustration. Datastructures are no exception to that maxim.

Returning to the task at hand, how might we use this dictionary? If we had this dictionary available on interactive Python, we could access the petitioner using the standard dictionary indexing notation.

```
>>> miranda['petitioner']
'miranda'
>>> miranda['respondent']
'arizona'
```

Just as with any other dictionary, place the key in square brackets following the name of the dictionary to access the corresponding value. We could use these values to create larger expressions.

```
>>> miranda['petitioner'] + ' v. ' + miranda['respondent']
'miranda v. arizona'
```

We can access any dictionary value in a similar manner.

```
>>> miranda['syllabus']
'In each of these cases, the defendant, while in police custody ...'
```

Likewise, we can access the list of justices who voted for the petitioner with the `'for_petitioner'` key.

```
>>> miranda['for_petitioner']
['warren', 'black', 'douglas', 'brennan', 'fortas']
```

Once again, we can use these values in an expression.

```
>>> for = len(miranda['for_petitioner'])
>>> against = len(miranda['for_respondent'])
>>> 'Decided: {0}-{1}'.format(for, against)
'Decided: 5-4'
```

By taking the length of the list of votes for the petitioner and for the respondent (the first two commands), we can create a string that describes the number of votes each side received.

We could use a for-loop to access the names of the justices who voted for each side.

```
26 # Print the votes for the petitioner.
27 print('Votes for ' + miranda['petitioner'])
28 for justice in miranda['for_petitioner']:
29     print('* ' + justice)
30
31 # Print an empty line.
32 print()
```

```
33
34 # Print the votes for the respondent.
35 print('Votes for ' + miranda['respondent'])
36 for justice in miranda['for_respondent']:
37     print('* ' + justice)
```

Lines 27-29 iterate through the list of votes for the petitioner and print a bulleted list of its entries. Lines 35-37 do the same for the respondent. This program will print the output below:

```
Votes for miranda
* warren
* black
* douglas
* brennan
* fortas

Votes for arizona
* harlan
* stewart
* white
* clark
```

Finally, we can also manipulate the datastructure under the key 'opinions'. Doing so requires us to index multiple times. To refer to Justice Harlan's dissent, for example, we will need to index the dictionary miranda with the key 'opinions' to get the dictionary of opinions.

```
>>> opinions_dict = miranda['opinions']
```

We will then need to index this dictionary with the key 'harlan' to access the nested dictionary representing Justice Harlan's opinion.

```
>>> harlan_dict = opinions_dict['harlan']
```

The variable harlan_dict now contains the dictionary on lines 16-19 of the original miranda dictionary. Finally, we can index this dictionary with the key 'text' to get the text of the opinion.

```
>>> harlan_dict['text']
'I believe that the decision of the Court represents poor ...'
```

Rather than write each of these intermediate indexing steps, we can condense all of this indexing into one Python expression.

```
>>> miranda['opinions']['harlan']['text']
'I believe that the decision of the Court represents poor ...'
```

Starting with the original miranda dictionary, the first indexing operation (the leftmost square brackets with the key 'opinions') accesses the nested dictionary of opinions (lines 7-24 of the program that created the miranda variable). Once

we have this dictionary, the second indexing operation (the middle square brackets with the key 'harlan') access Justice Harlan's dictionary nested within the 'opinions' dictionary (lines 16-19). Finally, the rightmost set of square brackets (with the key 'text') indexes Justice Harlan's dictionary, getting the opinion string itself.

Another way to understand this repeated indexing is to consider the way Python evaluates expressions. First, Python evaluates the variable miranda , arriving at the value of the underlying dictionary (the entirety of lines 1-24). It then evaluates the first indexing operation (the leftmost set of square brackets applied to this dictionary), arriving at the value of the 'opinions' dictionary. Since it finished with the leftmost indexing operation, it proceeds rightward, evaluating the middle indexing operation (with the key 'harlan') on the 'opinions' dictionary to arrive at the dictionary containing Justice Harlan's opinion dictionary. Finally, Python evaluates the last indexing operation (the rightmost set of square brackets) applied to Justice Harlan's dictionary, arriving at the text of the opinion.

We could use this same style of indexing to access the name of Justice Black from the list of votes for the petitioner

```
miranda['for_petitioner '][1]
```

Justice White from the list of votes for the respondent

```
miranda['for_respondent'][2]
```

the type of opinion Justice White issued

```
miranda['opinions '][' white '][' type']
```

or the text of Justice Clark's opinion

```
miranda['opinions '][' clark '][' text ']
```

Looking back over the past few pages, we have assembled a datastructure that made it possible to represent and efficiently explore Supreme Court cases. Doing so was no small undertaking—it required three levels of nested dictionaries. Now that we have a datastructure for representing individual Supreme Court cases, we are going to use it as a building block to construct even larger datastructures over the following sections.

10.5.2 All Supreme Court Cases

Now that we have a way to represent one Supreme Court case, we will build on that accomplishment to pursue a more ambitious goal: representing *all* Supreme Court cases that have ever been argued. Although this task may sound monumental at first glance, it is actually just a small variation on the job we just completed. At its very simplest, this datastructure could ental simply building a list of individual cases assembled in the style of the previous section.

```
all_cases = [miranda, brown_v_board, roe_v_wade, dred_scott, ...]
```

However, this choice of datastructure leaves much to be desired. In order to access *Brown v. Board of Education*, one would need to somehow know beforehand that it was at index 1. As we have done many times throughout this chapter, it is worth asking how we expect to try to find Supreme Court cases in the future. Answering this question will guide us toward designing datastructure that is useful and efficient in practice.

Every Supreme Court case has a string that specifically identifies it—its citation. For example, Miranda v. Arizona's citation is '384 US 436'. In legal documents, this string is enough to uniquely refer to this case in particular. One use of our datastructure for managing Supreme Court cases could be to look up cases by their citations. This insight suggests that we should use a dictionary where each key is a string containing a citation and each value is the corresponding case (as represented with the datastructure we created in the previous section. Concretely, our datastructure would look like the following:

```
all_cases = {
  '384 US 436' : miranda,
  '347 US 483' : brown_v_board,
  '410 US 113' : roe_v_wade,
  '60 US 393' : dred_scott,
   ...
}
```

Once we had this dictionary, we could use the standard indexing operation to access individual cases.

```
>>> all_cases['384 US 436']
{ 'respondent': 'arizona', 'for_petitioner': ['warren', ...
```

If we wanted to access Justice Warren's majority opinion in the case, we would need to:

1. Index the all_cases dictionary with the key '384 US 436' to access the Miranda dictionary.
2. Index the Miranda dictionary with the key 'opinions' to get the dictionary of opinions in the caes.
3. Index the opinions dictionary with the key 'warren' to get Justice Warren's opinion dictionary.
4. Index Justice Warren's opinion dictionary with the key 'text' to get the text of his opinion.

Concretely:

```
>>> all_cases['384 US 436']['opinions']['warren']['text']
'The cases before us raise questions which go to the roots ...'
```

This completes the task of designing a datastructure to manage all Supreme Court cases ever argued. Looking back, the hardest work was in designing

the right datastructure for managing an indivdual Supreme Court opinion and case. In this section, we simply built on that effort by adding one higher level—another dictionary to manage a collection of cases (making four levels of nested dictionaries so far).

10.5.3 Reference Network of Supreme Court Opinions

When we first set out to represent Supreme Court cases in Python, we observed that there are numerous different qualities of Supreme Court cases that we could include in our datastructure. For the sake of brevity, we chose only a few particularly illustrative examples (the petitioner, respondent, opinions, etc.). In this section, we will add one additional trait to these Supreme Court case datastructures, and—in doing so—enable a range of new analyses.

In the precedent-based American common law system, new court decisions are informed by other court decisions made in the past. As such, Supreme Court opinions are full of references to other opinions. In this section, we will augment our datastructure to include these inter-opinion references, making it possible to analyze the reference network between cases. Which opinions tend to cite which others? Can we identify topic areas or influential opinions by looking at reference information alone? The PageRank algorithm that made the Google search engine famous determines which websites are most "relevant" to a particular search solely using the reference network between web pages. Although we will leave these sophisticated analyses for later in your programming career, the first step toward making it possible to use these techniques is to built a datastructure that contains this information.

To augment our existing datastructure with reference information, we will need to answer two questions that should now begin to look familiar:

1. How and where do we store the references?
2. How do we represent a reference?

Let's start with the first question. Each opinion has its own separate set of references, so it seems logical that the references should be added to the dictionaries used to represent each opinion. In other words, the dictionaries that store the `'type'` of each opinion and the `'text'` of each opinion should all store the `'references'` made by each opinion. For example, we should be able to write

```
all_cases['384 US 436']['opinions']['warren']['references']
```

to access the references made by Justice Warren's opinion in 384 US 436 (i.e., Miranda).

How should the references be organized? Should they take the form of a list or a dictionary? The references are simply a collection of other opinions that this opinion cites. There are no natural keys or ordering—just a collection of values. Considering these loose constraints, a list should suffice for storing the references.

Finally, how do we represent a reference? A reference points to a particular opinion of another case. In the datastructure we have established so far, cases are identified by their citations (e.g., 384 US 436) and opinions are identified by the justice who authored them. These two pieces of information will therefore be sufficient to identify the opinion that a reference points to. There are many ways that we could represent these two fields. We could use a dictionary (e.g., `{ 'case' : '384 US 436', 'author' : 'harlan'}`) or a tuple with two elements (e.g., ('384 US 436', 'harlan')). Either of these choices are justifiable, but the dictionary option is more explicit and will likely be clearer to readers.

In summary, each reference is a dictionary with two elements: the case being cited and the author of the opinion being cited. Each dictionary representing an opinion has a list of these references representing all of the other Supreme Court opinions that its underlying opinion cites. Concretely, the Justice Warren's opinion in Miranda would take the following form:

```
{
  'type' : 'majority',
  'text' : 'The cases before us raise questions which go to the roots ...'
  'references' : [
      {'case' : '161 US 591', 'author' : 'brown'},
      {'case' : '217 US 349', 'author' : 'mckenna'},
      {'case' : '251 US 385', 'author' : 'holmes'},
      {'case' : '378 US 478', 'author' : 'goldberg'},
      ...
  ]
}
```

As before, the opinion has a key for the type of opinion and for the text of the opinion. In this section, we added the key for `'references'`. The references take the form of a list of dictionaries, each of which specifies the case and author of the opinion referenced. Below, we summarize where this component fits into our larger datastructure for managing Supreme Court case.

- We store all of the Supreme Court cases in a dictionary that maps each case's citation (e.g., '384 US 436') to the datastructure representing the case itself.

- We store each case in a dictionary that contains several pieces of information about the case (e.g., petitioner, respondent, opinions, etc.).

- Within a case's dictionary, we store the opinions in a nested dictionary. Each key of this nested dictionary is the name of the justice who wrote the opinion, and each value is a datastructure representing the opinion itself.

- Each opinion is a dictionary as portrayed in the example above. It has keys for the type of opinion, the text of the opinion, and the references the opinion makes to other Supreme Court opinions.

- The references are stored as a list of dictionaries, where each dictionary represents a single reference.
- Each reference is stored as a dictionary with two key-value pairs containing the case and author of the opinion being referenced.

In other words, a citation is a dictionary nested inside a list (all citations) nested inside a dictionary (an opinion) nested inside a dictionary (all opinions for a case) nested inside a dictionary (a case) nested inside a dictionary (all cases). All in all, six levels of nesting.

We will close by writing a simple program that takes advantage of this infrastructure. It will take as input a case citation and an opinion author, and it will print the names of all of the cases referenced by this opinion.

First, we will create the dictionary containing all Supreme Court cases. We will assume that this dictionary has been stored in a JSON file called supreme_court_cases.json.

```
1  import json
2  all_cases = json.loads(open('supreme_court_cases.json'))
```

Next, we need to read as input the opinion whose references we are to print.

```
4  # Read the case and opinion author as input.
5  case_cite = input('Case: ').upper().strip()
6  case_author = input('Opinion author: ').lower().strip()
```

After reading these values as input, we perform a small amount of data cleaning. We convert the case citation to all uppercase (converting '384 us 436' to '384 US 436') and strip off any extra whitespace before or after the citation. Similarly convert the author's name to all lowercase and strip off any extraneous whitespace. Doing so makes our program a little more tolerant to a careless user.

Before proceeding any further with our program, we should validate that this case and opinion actually exist. Otherwise, we might accidentally try to access invalid dictionary keys and crash the program.

```
8  # Ensure that the opinion actually exists.
9  if case_cite not in all_cases:
10     print('No such case.')
11 elif case_author not in all_cases[case_cite]['opinions']:
12     print('No such opinion')
```

The if-statement on line 9 checks whether the name of the case is a valid key in the all_keys dictionary. If not, it prints an error message. If this test passes (meaning the case really does exist), the elif-statement on line 11 checks whether the case contains an opinion under the author's name. A different error message will print if the author is not found. Notice that we need to perform the checks in this order. If we performed them the other way around, we would first see whether the opinion existed, then whether the case existed. The problem with this logic is that, if the case didn't exist, the attempt at checking for the opinion

would result in a KeyError, since all_cases[case] would not be found. In short, when checking for the existence of keys in nested dictionaries, it is essential to start checking at outer levels of the dictionary before checking at inner ones.

If these checks pass, the substance of the program begins inside the else-statement.

```
14 # The opinion exists. Print all of the citations.
15 else:
16     # Retrieve the opinion datastructure.
17     opinion = all_cases[case_cite]['opinions'][author]
18
19     # For each citation...
20     for reference in opinion['references']:
21         # Retrieve the case referenced.
22         case = all_cases[reference['case']]
23
24         # Assemble the name of the case.
25         name = case['petitioner'] + ' v. ' + case['respondent']
26
27         # Print the case name and opinion author.
28         print('{0} ({1})'.format(name, reference['author']))
```

The remainder of the program involves a series of dictionary accesses to extract information from the datastructures we have created. On line 17, we extract the opinion datastructure for the opinion whose references we wish to explore. On line 20, we iterate through those references; recall that the references take the form of a list and each individual reference takes the form of a dictionary. On line 22, we access the citation string (e.g., '384 US 436') of the case that the citation dictionary references (the expression reference['case']); we then use that value as the key to retrieve the case itself from the all_cases dictionary. This case is stored in the variable case.

Line 25 assembles the name of the referenced case (e.g., 'miranda v. arizona ') and line 28 prints that name along with the author of the opinion. If we were to run this program on Justice Warren's opinion in Miranda v. Arizona (inputs '384 US 436' and 'warren'), we would see the following output:

```
brown v. walker (brown)
weems v. united states (mckenna)
silverthorne lumber co., inc. v. united states (holmes)
escobedo v. illinois (goldberg)
...
```

10.6 Summary

In some ways, this chapter covered very little new ground. The only new Python tool we introduced was an alternative way of storing data, a capability we

already possessed in the form of lists. At a higher level, however, this chapter has made it possible to take on dramatically bigger problems. It opened up the possibility of creating, managing, and interacting with large datasets that were previously far too cumbersome to tackle with lists alone. Dictionaries and lists are complementary tools that jointly manage to satisfy the needs of the vast majority of data-related tasks we will consider in this book. Although, technically-speaking, lists can replicate all of the same behavior of dictionaries, doing so would be so inefficient and unwieldy that it would make it nearly impossible to scale up to datasets like the complete repository of Supreme Court cases, the United States code, or a corpus of millions of comments on a proposed administrative rule.

Armed with both dictionaries and lists, we considered the more abstract but no less vital task of matching these datastructures to a large, complicated, real-world dataset. At each level, we considered whether lists or dictionaries best fit our data, made informed decisions, and slowly built bigger and bigger datastructures that represented everything from individual references to the complete repository of all Supreme Court opinions ever issued. Although the end result—with six levels of nested datastructures—may seem monumental, we built this edifice by asking the same simple questions again and again in a disciplined manner at each level, incrementally expanding the size of our creation step by step. This skill, although less concrete than learning how to iterate over a dictionary's keys, is as essential to good programming as knowing when to use a loop or a conditional.

This chapter concludes our study of Python's built-in types of data. Although we will spend many later chapters working with special-purpose types that extend the Python language with new capabilities like reading a PDF document, dictionaries, lists, tuples, integers, floats, strings, booleans, and the none type comprise a complete set of the types we intend to introduce from core Python language. In other words, you have all the tools you need to represent nearly any kind of data you will come across during your Python career.

Just as this chapter substantially expanded your ability organize data, the next chapter will do the same for your ability to organize programs themselves.

10.7 Cheatsheet

Dictionaries

Preliminaries

- Like a list, a dictionary is a type of data that stores zero or more other pieces of data. Each value that a dictionary stores is identified by a key. For example, in a dictionary that stores the sounds that animals make, the key `'cat'` identifies the value `'meow'` and the key `'pig'` identifies the value `'oink'`.
- To access or save a particular dictionary value, you need to know its key. Unlike lists, dictionaries have no notion of order between values that they store.
- Although Python permits you to use many different types as keys, we strongly recommend that you only use strings as keys.
- Within a dictionary, keys are unique. A key can only appear in a dictionary once.
- Dictionary values can be of any type. It is acceptable to use values of different types for different keys. For example, the key `'is_cat'` could have a boolean value (True or False) while the key `'name'` could have a string value (`'Pixie'` or `'Judge Posner'`).
- If two dictionaries are two different examples of the same type of data (i.e., two different Supreme Court cases), they should have the same schema.
- Dictionaries are reference types like lists.

Creating Dictionaries

A dictionary is created by enclosing zero or more key-value pairs (`'cat' : 'meow'` or `'pig' : 'oink'`) in curly brackets (`{}`) and separating them with commas. For example, the dictionary containing animal sounds would be written as

```
{'cat' : 'meow', 'pig' : 'oink'}
```

with keys `'cat'` and `'pig'` and their values `'meow'` and `'oink'` respectively.

Since dictionaries have no notion of order, the order in which the key-value pairs are written is of no consequence. The empty dictionary (a dictionary containing no items) is written as an empty pair of curly brackets `{}`.

Accessing Dictionary Elements

A value can be accessed from a dictionary by placing the corresponding key in square brackets following the name of the dictionary. This expression evaluates to the key's value.

```
>>> sounds = {'cat' : 'meow', 'pig' : 'oink' }
>>> sounds['cat']
'meow'
>>> sounds['pig']
'oink'
```

Modifying Dictionaries

A dictionary key can be set or modified to store a particular value by using the assignment (=) operator.

```
>>> sounds['dog'] = 'woof'
>>> sounds
{'cat' : 'meow', 'pig' : 'oink', 'dog' : 'woof'}
>>> sounds['cat'] = 'purr'
>>> sounds
{'cat' : 'purr','pig' : 'oink', 'dog' : 'woof'}
```

To the left of the = operator, write the name of the dictionary followed by square brackets containing the key. To the right of the = operator, write the corresponding value. If the key is already present, this new value will overwrite whichever value the dictionary currently stores.

To delete a key and its value from the dictionary, write the del keyword followed by a space and the name of the dictionary with the key to be deleted in square brackets.

```
>>> del sounds['cat']
>>> sounds
{'pig' : 'oink', 'dog' : 'woof'}
```

Working with Dictionaries

Length. The len function will evaluate to the number of values stored in a dictionary.

Testing for membership. The boolean in and not in operators test whether the key to the left of the operator is present in the dictionary to the right. If it is, the in operator evaluates to True; if it isn't, it evaluates to False. The not in operator has the opposite behavior.

```
>>> sounds
{'cat' : 'purr','pig' : 'oink', 'dog' : 'woof'}
>>> 'pig' in sounds
True
>>> 'cow' in sounds
False
```

Note that these operators check whether a *key* is present in the dictionary; it does not check for the presence of values.

```
>>> 'woof' in sounds
False
```

The in operator is far more efficient on dictionaries than on lists. If a list contains n values, then each use of the in operator will cause Python to examine $\frac{n}{2}$ elements of the list (on average). No matter how many keys a dictionary contains, the in operator will cause Python to examine no more than a couple of keys, meaning it will complete almost instantaneously.

Default values. The setdefault method sets the value of a key if the key isn't already present in the dictionary; if it is, then the existing value is not modified. The method operates on a dictionary object and takes two arguments: a key and a value.

```
>>> sounds
{'cat' : 'purr', 'dog' : 'woof'}
>>> sounds.setdefault('cow', 'moo')
>>> sounds
{'cat' : 'purr', 'dog' : 'woof', 'cow' : 'moo'}
>>> sounds.setdefault('dog', 'bow wow')
>>> sounds
{'cat' : 'purr', 'dog' : 'woof', 'cow' : 'moo'}
```

The get method attempts to access the value corresponding to a particular key. If the key is found, the method call evaluates to the corresponding value. Otherwise, it evaluates to a default value provided to the function. The method operates on a dictionary object and takes two arguments: the key to be accessed and the default value.

```
>>> sounds
{'cat' : 'purr', 'pig' : 'oink', 'dog' : 'woof'}
>>> sounds.get('dog', 'bow wow')
'woof'
>>> sounds.get('cow', 'moo')
'moo'
```

The get method does not modify its dictionary object.

Converting to a list. The keys method produces a list-like type containing the dictionary's keys in an arbitrary order. It must be passed to the list function to convert it into a list.

```
>>> list(sounds.keys())
['dog', 'pig', 'cat']
```

The values method produces a list-like type containing the dictionary's values in the same order as the keys method.

```
>>> list(sounds.values())
['woof', 'oink', 'purr']
```

The items method produces a list-like type containing key-value tuples of the dictionary's elements.

```
>>> list(sounds.items())
[('dog', 'woof'), ('pig', 'oink'), ('cat', 'purr')]
```

You can iterate over these lists using for-loops.

Deduplication. You can use a dictionary to remove the duplicates from a list of elements. Add the elements to a dictionary as the keys. Since a dictionary cannot contain duplicate keys, the dictionary's keys will contain the deduplicated collection of elements. You can turn this back into a list using the keys method.

JSON

JSON is a way of representing in string form any datastructure constructed from nested lists, nested dictionaries, and the five fundamental types we introduced in Chapter 3 (integers, floats, strings, booleans, and the none type).

Python comes with a library called json that provides a function for converting a Python datastructure into a JSON string (json.dumps) and a JSON string into a Python datastructure (json.loads). Before you can use these functions, you need to insert the statement

```
import json
```

Saving a Python datastructure to a JSON file. Assume that the datastructure is stored in the variable datastructure and we want to save the datastructure to the file datastructure.json.

1. Convert the datastructure into a JSON string.

   ```
   >>> datastructure_json = json.dumps(datastructure)
   ```

2. Write the string datastructure_json to a text file called datastructure.json.

   ```
   >>> fh = open('datastructure.json', 'w'
   >>> fh.write(datastructure_json)
   >>> fh.close()
   ```

Loading a Python datastructure from a JSON file. Assume that the JSON version datastructure is stored in the file datastructure.json and we want to restore the Python version to the variable datastructure.

1. Open the file in read-mode, read the contents of the JSON string, and close the file.

   ```
   >>> fh = open('datastructure.json')
   >>> datastructure_json = fh.read()
   >>> fh.close()
   ```

2. Reconstitute the Python datastructure.

   ```
   >>> datastructure = json.loads(datastructure_json)
   ```

Datastructures

A *datastructure* is a combination of nested lists, dictionaries, and other types used to represent some dataset (like a Supreme Court case or the United States Code). To design a datastructure, start at the very highest level and break your datastructure down into smaller pieces. Determine whether each piece should be represented as a list, a dictionary, or another type. If each of these pieces have smaller subcomponents, repeat the process for these smaller elements using nested lists, dictionaries, or other types. As you choose the appropriate representation for your dataset, consider how you intend to use it in your program and aim to make doing so as easy as possible.

Keywords

Accessing—The act of obtaining a value from a dictionary using the requisite key.

Datastructure—A combination of nested lists, dictionaries, and other types used to represent a dataset.

Data cleaning—The process of refining data from an outside source to remove inconsistencies and put it in a format such that it can be processed easily by a program.

Deduplication—The process of removing all duplicates from a collection of items.

Deserialize—To reconstitute a Python datastructure from a string. The inverse of serializing. Example: `json.loads` deserializes JSON strings into Python datastructures.

Eagerly—Performing an operation proactively.

Efficiency—The resources necessary for a program to complete its task. Efficiency is generally measured in the number of operations your program has to perform (which equates with the time necessary for the program to run) and the amount of space necessary to perform those operations.

Indexing—In the context of dictionaries, see: accessing.

Key—In the context of a dictionary, the name used to refer to a specific value that a dictionary stores.

Lazily—Performing an operation reactively at the last possible moment.

Library—A bundle of related Python functions that your program can import and call.

List-like type—A special Python type that is different from a list but still contains an ordered collection of elements and supports some of the same operations as lists. It can be converted into a list by passing it to the `list` function.

Schema—A dictionary's keys and the types of the corresponding values. This information is sufficient to write a program that operates on any dictionary configured in this fashion.

Serialize—To convert a Python datastructure of any kind into a string (or any other format where it can be saved to a file). Example: `json.dumps` serializes Python datastructures into JSON strings.

Set—A collection of items that contains no duplicates. You can mimic a set using a dictionary's keys.

Value—In the context of a dictionary, the element that a dictionary stores. Each value is referred to by a unique key.

Chapter 11

Functions and Modules

At this point, you have learned nearly all of the core features of Python. You know how to use conditionals and loops to shape the behavior of programs that manipulate sophisticated datastructures consisting of nested lists and dictionaries.

Although you are able to write big programs that accomplish substantial amounts of work, doing so can quickly become unwieldy. Past a certain point, programs become too large and complex to keep track of: you simply can't keep every detail of hundreds or thousands of labyrinthine lines of code in your head. This problem only becomes worse when you revisit a program days, weeks, or months later.

These challenges are especially acute when you are trying to write programs as a team. A big program is difficult, if not impossible, to divide into independent units of work that multiple people can work on in parallel.

Effective style and copious commenting do help, but this chapter will explore a far more robust, structural solution that Python provides precisely to help you manage large programs: functions. Functions make it possible to break large programs down into smaller, more manageable pieces.

Functions are already a vital part of our Python lexicon—we have been using them since the very first Python programs we studied in Chapter 4. For example, we have used the print function to produce output and the len function to measure the length of lists and strings. We have mentioned dozens of other functions in passing throughout our discussions of lists, strings, and dictionaries.

In this chapter, we will transition from mere users of functions to creators. Functions make it possible to add new features to the Python language that are customized for whatever task is at hand. We will learn to design custom functions that meet the specific needs of individual programs and make it possible to reuse valuable code across many programs. In doing so, we will come to understand how functions can make it easier to manage programs of arbitrary size and complexity. Like dictionaries, functions are not—strictly speaking—required to write large programs, but they confer so many benefits that they are indispensable in practice.

Functions are powerful tools, but using them properly requires a great deal of care. We will spend a large portion of this chapter rigorously delving into the inner-workings of functions in great detail. To motivate and justify this discussion, we hope that you will keep in mind the new capabilities you are about to unlock.

Making problems smaller. Functions allow you to break your code down into smaller units. Like len or print, each of these units has a clear, discrete purpose. By breaking down a big problem into smaller pieces, you make it easier to individually focus on the simple components that collectively come together to serve a larger purpose. Rather than trying to solve one big problem with a tangle of logic and code, you solve numerous smaller problems one at a time.

Abstraction. Functions allow you to divide your program into multiple levels of concerns. You can write functions to hide away the low-level details of manipulating a datastructure. This layer of low-level functions creates a new set of operations that higher-level functions can call. Your higher-level functions need not worry about the low-level details, since they have been hidden away by other functions.

Debugging. When a program is divided into pieces, you can focus your debugging energy on each individual component. As you have no doubt already discovered, smaller, simpler programs are easier to debug. Rather than trying to debug a massive, monolithic program with a multitude of corners where errors could hide, you can debug each function separately.

Code reuse. Functions make it possible to write a self-contained block of code once, give it a name, and reuse it as many times as you like. Imagine how you would manage to write any program at all if you had to copy and paste the code underlying the len or print functions every time you wanted to call them.

Isolation. A function is a self-contained world separate from the rest of your program. The program needs to know only how to call a function and the work a function promises to accomplish once called. The program doesn't need to know *how* a function works—in fact, it's often better that it doesn't. If you treat functions as black boxes whose inner workings do not need to be understood, you will have fewer details to keep in your head. As long as the print function always behaves correctly, should you really care how it works internally? When it comes to functions, ignorance is bliss: as long as a function works properly, you need not about how it manages to do so—just how to interact with it.

11.1 Creating Functions

By now, you are already an experienced user of functions. In this section, we will formalize some of the terminology we have used casually until now and, with this knowledge in hand, learn to design our own functions.

Each function is simply another Python program that you can execute by calling it. For example, typing the statement print() causes Python to momentarily pause executing your program to execute the code associated with the print function. This code, known as the function's body, appears somewhere on your computer; where, exactly, is not particularly important, and it is often better not to know. The body of every Python function you have ever used (including print, input, and max) is stored somewhere on your computer in this fashion. When the function's body has finished executing, returns to the place where the function was called and continues executing normally. After your call to print finishes and returns, Python will continue to the next line of your program normally.

Function calls can be embedded in other function calls. It's possible that the print function needs to call other functions to do its job (in fact, it's very likely that this is the case). If print makes another function call, Python will jump to that function's body, execute the code it finds there, and return, picking up where it left off in print. When print finishes, it will return execution to your program. These nested function calls are very common in Python; most functions you write will need to, themselves, call other functions.

Now that you have a general sense for the way execution flows through functions, we will are prepared to discuss the detailed mechanics behind this process.

11.1.1 Terminology

Names. Every function has a name, such as print or max. Python treats these names exactly as it does variable names. This means that you can create new variables that copy the behavior of existing functions.

```
>>> num_list = [3, 8, 4, 4, 1, 7]
>>> len(num_list)
6
>>> length_of_list = len
>>> length_of_list(num_list)
6
```

The first two commands at iteractive Python should be quite familiar. They created a list of six numbers (num_list) and evaluated its length using the len function. The third command takes advantage of the fact that function names are simply variables. We can describe its behavior in two ways:

1. In terms of variables, it creates a new variable, length_of_list, that copies the contents of the len variable.

2. In terms of functions, it creates a new function, length_of_list(), that behaves in exactly the same way as the len() function.

In Python, these two explanations mean exactly the same thing. The fourth command proves that the code behaves as we expect, showing that length_of_list can now be used just like len.

If we can copy functions from one variable to another, can we also overwrite those variables? In a word, yes.

```
>>> len = 15
>>> len
15
```

We have overwritten the value of the len variable with the integer 15. In doing so, we can no longer use len to take the length of lists and strings!

```
>>> len('Form follows function')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Python produces a TypeError stating that an 'int' object is not callable. In other words, len is now an integer (namely, 15) and Python will no longer let us call it like a function. This example is not a suggestion—it is a cautionary tale: many bugs originate by accidentally creating a variable with the same name as a function and then later trying to call that function. In the process, the function is overwritten and a similar error appears.

Function calls and arguments. When we want to use a function, we call it by following the name of the function with parentheses that enclose a list of zero or more arguments. For example:

- We call the input function by writing input(); input() doesn't require any arguments, although we can optionally pass it a single argument (input ('Please enter your name: ')) that causes it to print a prompt before collecting input from a user.

- We call the len function by writing len('hello'), len([1, 2, 3]), or len(name_of_list_or_string) to respectively compute the length of a string, a list, or a variable storing one of the two. The len function always requires a single argument, which is placed between the parentheses when it is called. If we were to call len without any arguments, we would see the following error:

  ```
  >>> len()
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: len() takes exactly one argument (0 given)
  ```

(Be sure to quit and reopen your interactive Python session if you overwrote the value of len earlier.) The error is self-explanatory: len always needs exactly one argument.

- We call the max function to compute the maximum value of several numbers by writing max(12) or max(1, 2, 3, 4). The max function can take a single argument, but the result is rather boring: max(12) will always be 12 since it is the only number present. We can also provide max with multiple arguments separated by commas; calling max in this fashion will cause the function call to evaluate to the biggest of its arguments.

When you call a function with arguments, those values are passed to the corresponding function body, where they are used to compute the function's eventual result. We will soon see the exact mechanics of how they are used in more detail. The arguments *must* be passed in the correct order. A function will ascribe different purposes to each argument in order; if arguments are provided in the wrong order, the function will use them in the wrong places and potentially produce an erroneous result or completely crash the program. Although the − operator is not a function, consider what would happen if we performed the calculation 3 − 1 rather than 1 − 3. Mixing up the order of function arguments can have similarly disastrous effects.

Value and reference arguments. It is useful to distinguish between two different types of arguments: those that are passed by value and those that are passed by reference. You may recall a similar discussion when we explored how integers and lists are copied between variables.

We briefly review that discussion before explaining how it applies to functions.

Suppose we execute the following commands on interactive Python:

```
>>> x = 5
>>> y = x
>>> y = 7
>>> x
5
```

We create an integer variable x that stores the value 5. We then set y equal to x. Finally, we modify y, setting it to 7. We find that, even after modifying y, x remains unchanged. The reason? The second command (y = x) copied the value of x to a separate location reserved for y. When we modified y (y = 7), we only modified y's storage area, leaving x's untouched. Python types that are copied in this way (integers, floating point numbers, booleans, and strings) are said to be copied by value.

We later discovered that other types of Python data, like lists, dictionaries, and most other new types we will encounter throughout the following chapters, are passed by reference. Consider the example below.

```
>>> x = [1, 2, 3]
```

```
>>> y = x
>>> y[1] = 100
>>> x
[1, 100, 3]
```

The example is identical to the previous one, except that we store lists rather than integers. The first command creates a list and stores a reference to that list in x. That reference is copied to y in the second command. This means that x and y refer to the same list, so modifying y also modifies x.

The same rules apply to value and reference types that are passed as arguments to functions.

When a variable storing an integer (or any other value type) is passed as an argument to a function, the value of that variable is copied when the function executes, meaning that the function cannot modify the value of the variable. When a variable storing a list, dictionary, or any other reference type is passed as an argument to a function, a reference to that value is copied when the function executes, meaning that the function *can* modify the value of the variable. This distinction is subtle, but it is important to keep in mind when creating and using functions.

Return values. A function call is just another Python expression. Just as the expression `2 + 2` evaluates to the integer 4, the expression `len('hello')` evaluates to the integer 5 and the expression `sorted([2, 1, 5, 8, 3])` evaluates to the list `[1, 2, 3, 5, 8]`. We can then use these values just as we would any other Python value, saving them to variables, printing them, or situating them inside larger computations.

The value to which a function call evaluates is known as its return value. When we explore the process of designing functions, the rationale behind this seemingly cryptic name will become apparent. Many of the functions we have encountered so far produce return values. For example, the input function evaluates to (returns) the string that the user typed as input. The len function returns the length of a list or string. The sorted function returns the sorted equivalent of a list passed as its argument. The max function returns the largest of its numerical arguments.

Effects. Not all functions have return values. Consider how we typically call the print function: we call it on a line by itself and we do not save the result of the call to a variable. Were we to do so, we would find that it returns the value None, i.e., nothing.

The print function still does *something*—it prints a string as output. We refer to this printing as an effect—an impact that a function had on the world that cannot be classified as a return value. Other effects might include modifying an argument (if that argument is a reference type), manipulating files on a hard drive, or communicating with a website over the Internet.

Occasionally, a function will have both an effect and a return value. The input function, for example, can both print a string to the screen and return

the text that the user entered.

When a function has no effects, it is said to be pure, meaning that its arguments and return value are enough information to fully describe how the function works. When a function has effects, it is correspondingly impure, meaning that its description will need to include effects like printing or modifying arguments. A function that has neither effects nor a return value accomplishes no work—it is useless, and calling it is a waste of electricity.

Function signatures. When trying to call a function, there are a few vital pieces of information that are necessary to use it properly.

1. We need to know its name.
2. We need to know how many arguments it requires.
3. We need to know the order in which it expects its arguments.
4. We need to know the types of arguments that it expects. We cannot pass an integer where the function expects a string or a string where it expects a list of strings. Failing to provide the expect types of data can create a variety of subtle, hard-to-debug errors.
5. We need to know the return value the function produces and its type.
6. We need to know whether the function modifies any of its arguments and how it does so.
7. We need to know any effects that it has.

Collectively, this information is known as a function signature—a complete description of its behavior. Observe that we do not need to know *how* the function manages to perform the promised task so long as it does so faithfully every time we call it. The creators of Python could swap in a different underlying implementation of the print function tomorrow but, as long as the function signature remains the same, all of our code will continue to work and we will never notice the change. When calling functions, we care only about behavior, not about the underlying mechanisms at work.

A function signature is a contract with a user. The function promises that, so long as a user provides the proper arguments in the proper places, the function will produce a specific set of effects and the proper return value. Outside of the parameters of the contract, the function is free to do as it pleases to meet its obligations.

11.1.2 Defining Functions

Basics. Now that we are expert users of functions, we are ready to create our own. We will begin with a simple function, one that takes no arguments, prints the string `'Form follows function'`, and has no return value.

```
1 def print_quote():
2     print('Form follows function')
```

Every function begins with the def keyword. Much like the if, while, and for keywords, def tells Python that a function declaration is about to begin. After the def keyword is a blank space followed by the name of the function. In this case, we have named our function print_quote. This naming process is just like creating a new variable.

After the name of the function is a set of parentheses. Right now the parentheses are empty, since the function takes no arguments. Later, we will see how to use these parentheses to build functions that do accept arguments.

The opening line of a function closes with a colon.

After the first line of a function (the one beginning with def) is an indented code block. Just as with if-statements and loops, this code block contains all of the code that executes when the function is called; it is known as the function's body. The body of the function we have defined above contains a single line of code that prints the string, 'Form follows function'.

Now that we have defined a function, we can call it.

```
1 def print_quote():
2     print('Form follows function')
3
4 print_quote()
```

We do so just as we would with the print, input, or len functions: we write the name of the function followed by parentheses containing its arguments; since print_quote does not accept any arguments, we do not provide any arguments between the parentheses when we call it. The result of executing the program appears below.

```
Form follows function
```

The program runs from top to bottom. On line 1, Python sees that a new function is being declared. It creates a variable with the name print_quote and stores in the variable the function defined on lines 1 and 2—one that takes no arguments and contains as its body the lone print statement on line 2. On line 4, we call that function. Execution immediately jumps to the body of the function (line 2), where it prints the string 'Form follows function'. When the function body runs out of code, execution returns to the line where the function was called (line 4). With no more code to execute, the program terminates.

Before we continue, observe a few important features of this program.

1. The function declaration on lines 1 and 2 did not print anything. Rather, it created a function with the *capability* to print and saved it to a variable. The function only printed output when it was called on line 4.

2. Functions have radically disrupted our understanding of the way Python programs are executed. Until this chapter, a program was executed from

top to bottom with minor, regimented deviations for while-loops and for-loops. Functions make it possible for program execution to jump back and forth across different regions of a program in an unconstrained and seemingly chaotic fashion. The function call on line 4 caused execution to briefly move to lines 1 and 2, after which it resumed from line 4, causing the program to terminate. That function might have called a different function elsewhere in the program that could further violate the linearity we have come to expect. (In fact, it does exactly that: on line 2, the print_quote function calls the print function, causing code somewhere else on your computer to execute.) These jumps back and forth are a fact of life in the kinds of programs you will write going forward, but it is important to explicitly note this paradigm shift.

Python cannot call a function until the function has been declared. Imagine if we modified our program so that we call print_quote before we define it.

```
1  print_quote()
2
3  def print_quote():
4      print('Form follows function')
```

Python would produce the following output:

```
Traceback (most recent call last):
  File "print_quote.py", line 1, in <module>
    print_quote()
NameError: name 'print_quote' is not defined
```

This is the same error that Python produces when it comes across a variable that does not exist. Python complains that the name print_quote, which is referred to on line 1, has not yet been declared at that point in the program.

Multiple function calls. As you have already experienced with print and len, you can call function more than once during the execution of a program. Consider another slight modification to the program below:

```
1  def print_quote():
2      print('Form follows function')
3
4  print('First time.')
5  print_quote()
6
7  print('Second time.')
8  print_quote()
9
10 print('Third time.')
11 print_quote()
12
13 print('All done!')
```

When executed, the program will produce the following output:

```
First time.
Form follows function
Second time.
Form follows function
Third time.
Form follows function
All done!
```

The first two lines of the program declare the function just as before. As always, the declaration simply creates the function and saves it to the variable print_quote; the function does not yet execute. The program continues executing from top to bottom. On line 4, it prints the string 'First time.' and proceeds to line 5. Line 5 calls the function print_quote, causing Python to jump to line 2 and execute the function body. Line 2 prints the string 'Form follows function'. After doing so, Python has exhausted all of the code in the function body, so execution picks up where it left off on line 5.

Since line 5 is done executing, Python continues moving from top to bottom, printing the string 'Second time.' on line 7. After doing so, Python executes the function call on line 8. Execution again jumps to the function body on line 2, prints the string 'Form follows function', and returns to line 8. This process repeats again: Python continues moving from top to bottom, printing 'Third time.' on line 10 and calling the function again on line 11. For the final time, execution jumps to line 2, prints 'Form follows function', and returns to line 11. Finally, execution continues to line 13, prints 'All done!', and terminates.

Although this summary is perhaps overly detailed, you should now have a good sense for the way functions affect the flow of execution.

Multiple functions. We could easily add a second function to the preceding program.

```
1  def print_author():
2      print('-- Louis Sullivan')
3
4  def print_quote():
5      print('Form follows function')
6
7  print_quote()
8  print_quote()
9  print_quote()
10
11 print_author()
```

The program should execute exactly as before with three minor changes.

1. We create two functions at the beginning rather than just one. On lines 1 and 2, we define the print_author function; on lines 4 and 5 we define the print_quote function.

2. The program calls the print_author function on line 11, causing execution to briefly jump to line 2 (the body of the print_author function), print the purported author of the quote, and return to line 11 to resume normal execution.

3. We remove extraneous print statements to make the function's behavior clearer.

The result of running this updated program appears below.

```
Form follows function.
Form follows function.
Form follows function.
-- Louis Sullivan
```

The first three printed lines correspond to thre three calls to print_quote on lines 7, 8, and 9. The final printed line corresponds to the call to print_author on line 11.

We consider one final modification to the program that contrasts with the previous approach. Specifically, we call a function from the body of another function.

```
1  def print_author():
2      print('-- Louis Sullivan')
3
4  def print_quote():
5      print('Form follows function')
6      print_author()
7
8  print_quote()
9  print_quote()
10 print_quote()
```

Rather than calling print_author once at the end of the program as before, we place the call to print_author *inside* of the body of print_quote (line 6). Let's explore how the program now behaves. Python begins executing from top to bottom, declaring the print_author and print_quote functions on lines 1-2 and 4-6 respectively. Python encounters the first function call on line 8. This call is to print_quote, causing execution to jump to the body of the function on line 5. Python first prints the string 'Form follows function'.

In this version of the program, the function body still contains more code to execute, so Python continues to line 6. Line 6 contains a nested function call to the print_author function. Execution immediately jumps again, this time to the body of print_author on line 2. Note that we are now two levels deep. The function call on line 8 is waiting for print_quote to finish executing; print_quote, in turn, has paused on line 6 while waiting for print_author to finish executing. On line 2, print_author prints the string, '-- Louis Sullivan'; since its body has no more code to execute, Python returns to the next level

up: line 6 in print_quote. Since this function body has also completed, Python returns to line 8 and continues executing.

On line 9, Python encounters another call to print_quote, so it jumps back to lines 5 and 6, printing 'Form follows function' and calling print_author. After the call to print_author finishes, execution returns to line 6 (the end of the body of print_quote) and then returns to line 9. This entire process repeats once more on line 10, after which the program terminates.

The output of the program appears below.

```
Form follows function.
-- Louis Sullivan
Form follows function.
-- Louis Sullivan
Form follows function.
-- Louis Sullivan
```

This output differs slightly from that produced by the previous program. The program calls print_quote three times and each call to print_quote leads to a nested call to print_author, meaning that print_author is also called three times. Thus, the quote's author is printed after each of the three times the quote itself is printed.

11.1.3 Arguments

The functions we have considered so far are rather boring. Using the terminology we introduced earlier, these functions are effectful (they print output to the terminal), but they do not take any arguments, meaning that they will always do exactly the same thing every time they are called.

To create a function with an argument, follow the same recipe as before with one addition: between the parentheses of the function declaration, write the name of a variable. When the function is called, the argument provided in the function call (the value placed between the parentheses of the function call) is assigned to the variable in the function declaration. This variable can then be used anywhere within the function body.

This explanation is much easier to understand with a concrete example.

```
1 def greet_name(name):
2     print('Hello, ' + name + '!')
```

Above, we have created a function with an argument. The function is called greet_name, and it takes a single string argument representing the name of a person who the function should greet. This argument is, aptly enough, given the title name. We can call the function in the following manner:

```
4 greet_name('Larry')
```

When Python calls the function, it jumps from line 4 (the function call) to line 1 (the beginning of the function). Before executing the function body on line 2, Python creates the variable specified in the function declaration (name) and

assigns to it the value passed in the function call ('Larry'). In other words, we can imagine that Python performs the assignment name = 'Larry' just before the function body is executed. When Python gets to line 2, the variable name has the value 'Larry' and the print statement will produce the following output:

```
Hello, Larry!
```

The variable name will be assigned to whatever value we fill in as the function call's argument. Were we to modify line 4 to

```
4 greet_name('Mary')
```

then Python would perform the assignment name = 'Mary' just before the function body executes, leading to the output

```
Hello, Mary!
```

The variable name is created fresh every time we call the function. If we call the function multiple times with several different names, it will print a greeting to each name as it is passed as an argument:

```
4 greet_name('Larry')
5 greet_name('Mary')
6 greet_name('Carrie')
```

```
Hello, Larry!
Hello, Mary!
Hello, Carrie!
```

As before, the first two lines of the program will create the greet_name function. When the program reaches line 4, execution will transfer from the function call to the body of greet_name. Before executing the body of the function, the variable name will be created with the value 'Larry' (the argument passed to this particular function call). The program will then print

```
Hello, Larry!
```

on line 2. Since the function body has no more code remaining, Python will return to line 4. On line 5, a similar process will occur. The function call will transfer execution back to line 2 with the variable name assigned to 'Mary' (the argument of the function call on line 5). The function will print a greeting to Mary, after which execution will return to line 5. Line 6 behaves similarly, assigning name to 'Carrie' and printing the greeting 'Hello, Carrie!' accordingly.

If the creation and recreation of the variable name seems spooky or mysterious, remember that we have seen this behavior elsewhere in Python. The variable of a for-loop is magically created and assigned to the next item in a list at the beginning of each loop iteration; every time this happens, the variable's old value is overwritten completely. A similar process is at play here: every time a function is called, the variables that receive its arguments are overwritten with the new values passed by the function call.

Before moving on, it is important to clarify three properties of the variables that receive function arguments.

1. We can choose any name we like for these variables. Rather than calling the variable name in the program above, we could refer to it as name_of_person or (if we neglected style entirely) albert. If we change the name of the variable, we also need to change the name in all places where the variable is used within the function:

```
1 def greet_name(name_of_person):
2     print('hello, ' + name_of_person)
```

   Notice that we changed the name of the variable both on line 1 (where it is declared) and on line 2 (where it is used). As with all variables, it is important to give descriptive names that clearly convey the variable's role in the function.

2. The variable exists only within the function's body (also known as the function's scope, a topic we will discuss soon in more depth). If we tried to use the variable name_of_person outside of the function greet_name

```
1 def greet_name(name_of_person):
2     print('hello, ' + name_of_person)
3
4 greet_name('Larry')
5 print(name_of_person)
```

   we would see the following output:

```
hello, Larry
Traceback (most recent call last):
  File "argument_example.py", line 5, in <module>
    print(name_of_person)
NameError: name 'name_of_person' is not defined
```

   The first line of the output contains the result of the function call on line 4. The program successfully jumps into the body of greet_name, prints a greeting for Larry, and returns to line 4. It is line 5 that causes the program to crash. The error message should look familiar—it's the same message that prints whenever we attempt to use a variable that has not yet been defined. Since greet_name's argument, name_of_person, exists only within the body of the function, Python does not recognize the name name_of_person outside of the body of the function (in this case, on line 6). As such, the program crashes and Python produces an error.

11.1.4 Multiple Arguments

We are free to write functions that take more than one argument. As you already know, you can pass a function multiple arguments by separating the

arguments with commas. For example, we could call the max function with five arguments as below:

```
max(3, 9, 2, 5, 8)
```

When creating a function with multiple arguments, we need to provide a corresponding set of variables ready to receive the values that are passed. For example, we might want to modify the program from the previous section to take a custom greeting along with a person's name:

```
1 def greet_name(greeting, name):
2     print(greeting + ', ' + name + '!')
```

The function now has two variables between the parentheses on line 1: greeting (to receive the greeting) and name (to receive the name of the person to be greeted just likse before). We would call the function as follows:

```
4 greet_name('Good morning', 'Jerry')
```

When Python executes the program, it begin on lines 1 and 2, creating the function and saving it to a variable called greet_name. It then proceeds to the function call on line 4. This call causes program execution to jump to the body of greet_name on line 2.

Before Python executes the body of greet_name, however, it creates the function's arguments. It assigns the first variable, greeting, the value of the first argument, 'Good morning'. Likewise, it assigns the second variable, name , the value of the second argument, 'Jerry'. To generalize, Python assigns the function's arguments to the corresponding variables in the order they are listed. The first argument always goes to greeting and the second to name. The program would have the following output:

```
Good morning, Jerry!
```

Were we to inadvertently swap the order of the arguments in our function call

```
4 greet_name('Jerry', 'Good morning')
```

Python would assign the first argument, 'Jerry', to the variable, greeting, and the second argument, 'Good morning', to the second variable, name. The output would be mangled accordingly:

```
Jerry, Good morning!
```

Although we have shown functions with only one or two arguments, you can repeat a similar process to create functions with as many arguments as desired. Simply declare a function containing in its parentheses a series of variables of the appropriate length (separated by commas) and call the function with the corresponding number of arguments (also separated by commas).

11.1.5 Modifying Arguments

Earlier, we discussed the difference between passing arguments by value and by reference. Now that we know how to create functions, we can revisit this topic in concrete terms.

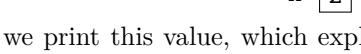
Value types. Imagine we have written the function below, which takes a single integer argument and attempts to increase its value by 1.

```
1  # A failed attempt to increase the value of a function's argument by 1.
2  def broken_increment(num):
3      num = num + 1
4      print('Value of num at end of function: ' + str(num))
5
6  # Create a variable and print its initial value.
7  x = 2
8  print('Value of x before calling the function: ' + str(x))
9
10 # Attempt to increase the value of the variable by 1.
11 broken_increment(x)
12
13 # Print the value of the variable after the function call.
14 print('Value of x after calling the function: ' + str(x))
```

On lines 1 through 4, we create a function called broken_increment. It is so named because it attempts to increase the value of the function's argument by 1 but, as we will soon see, fails to do so. It takes a single integer argument called num. In its body, it increases the value of num by 1 (line 3) and prints the value of num after doing so (line 4). The main body of the program begins on line 7, where we create a variable called x and initialize it to 2. We print the variable's value before and after passing it as the argument to broken_increment on line 11. The output of running this program is below:

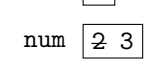
```
Value of x before calling the function: 2
Value of num at end of function: 3
Value of x after calling the function: 2
```

Upon close inspection, we find that the function did not affect x's value. It did, however, manage to increase the value of num, which exists only within the function, by 1. To get a better understanding for what exactly happened, let's examine the way Python represents the variables internally. On line 7, we create the variable x and assign it the value 2. Internally, Python creates space in your computer's memory for x and fills it with the number 2.



On line 8, we print this value, which explains the first line of output that the program produces. On line 11, we call broken_increment with x as an argument. Python immediately jumps to the body of broken_increment on line 3. Before executing the body of the function, however, Python creates the variable num, which receives the argument passed to broken_increment. The value of the argument—the value of x—is copied to the storage space Python sets aside for num.

Inside the function body on line 3, we then increase the value of num by 1.



Notice that, since x's storage space is separate from that of num, x's value remains unchanged. The print statement on line 4 prints the updated value of num, which explains the second line of output that the program produces. When the function body ends, execution returns to line 11 and continues to line 14. Python then prints the value of x, which is still 2. This explains the third line of output that the program produces.

The lesson from this example is that, when passing a value-type variable as an argument to a function, the function cannot change the value of the variable.

Reference types. Consider what would happen if we modified the previous program slightly to handle lists rather than integers.

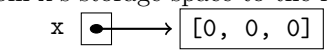
```
1  # Increases the value of the first item in the list comprising the function'
       s argument by 1.
2  def increment_first_item(nums):
3      nums[0] = nums[0] + 1
4      print('Value of nums at end of function: ' + str(nums))
5
6  # Create a variable and print its initial value.
7  x = [0, 0, 0]
8  print('Value of x before calling the function: ' + str(x))
9
10 # Attempt to increase the value of the first item in x by 1.
11 increment_first_item(x)
12
13 # Print the value of the variable after the function call.
14 print('Value of x after calling the function: ' + str(x))
```

This program is largely identical to the previous one. The function now takes a list as an argument, and it increases the value of the first item in the list by 1. The name of the function, the name of its argument, and several of the print statements and comments have been updated accordingly. The resulting output is below:

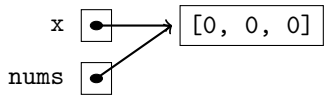
```
Value of x before calling the function: [0, 0, 0]
Value of nums at end of function: [1, 0, 0]
Value of x after calling the function: [1, 0, 0]
```

This time, the function seems to have successfully modified the argument—the change to the argument list seems to stick even after the function call has returned. To understand why this program failed on value types but seems to have worked on reference types, let's consider what Python is doing under the hood.

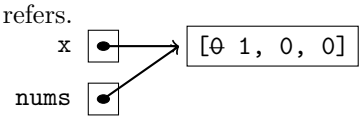
On line 7, we create a variable x and assign it the list [0, 0, 0]. As we discussed in Chapter 8, Python creates storage space for reference types in two steps. First, it creates the list itself. Next, it creates space for x and points a reference from x's storage space to the location of the list.



Unsurprisingly, the first line of output prints the initial value of x. When we call the increment_first_item function on line 11, execution jumps to the body of the function (line 2). As we have grown to expect, Python initializes the variable that receives the function's argument, nums, before executing the function body. Python sets aside storage space for nums and copies the *reference* from x's storage space to that of nums. In other words, x and nums now refer to the same list.



On line 3, when we modify the list to which nums refers, we also modify the list to which x refers.



We print the updated value of nums on line 4 (reflected in the second line of output). Program execution then returns to line 11 and continues to line 14, where the value of x is also printed. Since the function modified the list to which x refers, the third line of output reflects the update performed on line 3.

The lesson from this second example is that, when passing a reference-type variable as an argument to a function, the function can change the value of the item that the variable refers to.

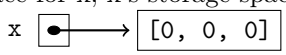
Modifying references. There are still limitations on the ways that functions can manipulate reference-type arguments. Consider one final change to the program. The only consequential modification is on line 3.

```
1  # Attempts (but fails) to overwrite the argument with the empty list.
2  def broken_set_to_empty_list(nums):
3      nums = []
4      print('Value of nums at end of function: ' + str(nums))
5
6  # Create a variable and print its initial value.
7  x = [0, 0, 0]
8  print('Value of x before calling the function: ' + str(x))
9
10 # Attempt to overwrite x with the empty list.
11 broken_set_to_empty_list(x)
12
13 # Print the value of the variable after the function call.
14 print('Value of x after calling the function: ' + str(x))
```

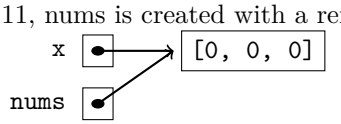
Rather than modifying an element of nums, we overwrite it entirely with the empty list. When we execute this program, we see the following output:

```
Value of x before calling the function: [0, 0, 0]
Value of nums at end of function: []
Value of x after calling the function: [0, 0, 0]
```

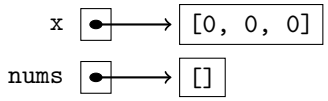
Perhaps surprisingly, the function was able to modify only nums—it failed to have any effect on x. If functions can modify reference-type variables, how do we explain this outcome? The previous rule about reference-type arguments contains an important caveat: the function can change only "the value of the item that the reference-type variable refers to". In other words, functions can modify elements of the list but cannot swap in a new list entirely. Let's consider this distinction concretely. Just as before, line 8 causes Python to create a list and storage space for x; x's storage space contains a reference to the list.



Likewise, when we enter the body of the function on line 3 due to the function call on line 11, nums is created with a reference to the same list as x.



Here is where this program differs from the previous example. On line 3, we ask Python to create a new list (namely, the empty list) and to overwrite nums with a reference to this new list. Visually:



This operation modifies nums' *reference*, not the list that the reference points to. The two variables now point to two separate lists and x is unchanged. The call to print on line 4 prints the current value of nums (the empty list), after which Python exits the function, returns to line 11, and continues to line 14. Since x has not been modified, Python prints the value [0, 0, 0].

Summary. We summarize these examples as follows:

1. When a value-type variable is passed as an argument to a function, the function cannot affect the value of the variable.
2. When a reference-type variable is passed as an argument to a function, the function can alter the item that the variable refers to.
3. When a reference-type variable is passed as an argument to a function, the function cannot change the reference itself. In other words, it cannot make the variable point to a completely different item.

The difference between items 2 and 3 is subtle, but investing a few minutes now to understand the way Python handles different kinds of arguments will likely save you hours of debugging over the course of your programming career.

11.1.6 Return Values

So far we have learned to create only effectful functions—functions like print that have an impact on the outside world (namely, printing output) but do not themselves evaluate to a value. How do we create functions like len, whose sole purpose is to generate a return value that can be stored or used in other expressions? To do so, we will need a new keyword: return.

Suppose we wanted to implement a function that determines the number of times that a particular element appears in a list. For example, we might ask how many times the integer 1 appears in the list [3, 1, 4, 1, 5, 9, 2] or the number of times the string 'argle−bargle' appears in a list of words from a Scalia opinion.

Before we try to write this function, we should describe its behavior—we should write its signature. The function will be named appearances and it will take two arguments. The first argument is a list and the second argument is the item we are trying to count in that list. These two arguments are sometimes referred to informally as haystack and needle respectively, since we are searching for the item needle in the list haystack (a reference to the expression about looking for a needle in a haystack). The function should not have any effects—it shouldn't print any output or modify its argument. When called, the function should evaluate to the number of appearances of needle in haystack. In other words, the function should behave as follows on the interactive Python:

```
>>> appearances([3, 1, 4, 1, 5, 9, 2], 1)
2
>>> opinion = 'whatever disappearing trail of its legalistic argle-bargle one
    chooses to follow'.split()
>>> appearances(opinion, 'argle-bargle')
1
```

Now that we have a clear idea of how our function should behave, we can begin to write it. Since we know the name of the function and the number, names, and order of its arguments, we have enough information to write the first line of the function declaration.

```
1 def appearances(haystack, needle):
```

What goes inside the function body? We will need a variable to keep track of the number of times needle was found in haystack. We will need to visit each element of haystack, test whether it is equal to needle, and—if it is—add one to our count. We already know how to implement this procedure with a for-loop:

```
1 def appearances(haystack, needle):
2     # Keep track of the appearances of needle.
3     count = 0
4
5     # Visit each element of haystack.
6     for element in haystack:
7         # Check whether element is needle.
```

```
8           if element == needle:
9               count = count + 1
```

The first line of the function declaration remains the same. On lines 2 and 3, we create a variable called count that keeps track of the number of times we have seen needle so far. It is initially set to 0. Line 5 begins a for-loop that iterates over each element in haystack. It contains a nested if-statement (line 7) that checks whether the current element is the same as needle; if so, it increases the value of count by 1 (line 8).

At the end of the function body, all that remains is to somehow pass the value of count back to the caller of the appearances function. We do so with a new Python keyword: return. We would use the keyword in our function as follows:

```
1  def appearances(haystack, needle):
2      # Keep track of the appearances of needle.
3      count = 0
4
5      # Visit each element of haystack.
6      for element in haystack:
7          # Check whether element is needle.
8          if element == needle:
9              count = count + 1
10
11     # Return the value of count.
12     return count
```

At the end of the function body we add a return statement. In English, line 12 says, "a call to this function should evaluate to the value of count." In other words, the expression following the keyword return becomes the value of the function call. If we were to add a call to appearances on line 14

```
14 print(appearances([3, 1, 4, 1, 5, 9, 2], 1))
```

then the program would print the output

```
2
```

which is the value of count at the end of this particular call to appearances.

You can put any expression you like after return. If we wanted calls to appearances to always evaluate to 27, we could write

```
12     return 27
```

Likewise, if we always wanted calls to appearances to evaluate to a string describing the count, we could substitute the following expression on line 12:

```
12     return 'Found {0} instances of {1} '.format(count, needle)
```

If we were to call the function with this new return statement using the arguments from line 14, then the call to appearances would evaluate to 'Found 2 instances of 1' and the print statement on line 14 would output

```
Found 2 instances of 1
```

The return keyword causes Python to immediately exit the function and return to the line of code from which the function was called. If we were to put any code after the return statement, it would be ignored completely.

```
1  def appearances(haystack, needle):
2      # Keep track of the appearances of needle.
3      count = 0
4
5      # Visit each element of haystack.
6      for element in haystack:
7          # Check whether element is needle.
8          if element == needle:
9              count = count + 1
10
11     # Return the value of count.
12     return count
13
14     print('Output that is within the function body but after the return.')
15     print('Python never reaches either of these print statements.')
16
17 print(appearances([3, 1, 4, 1, 5, 9, 2], 1))
```

The program above will still only print the output 2. The print statements on lines 14 and 15 will never execute.

The return keyword can appear in a function multiple times. In fact, this is a common strategy in functions that contain if-statements. Suppose we were trying to write a function that determines whether two integers will divide cleanly, leaving no decimals or remainder. The function will be called divides_cleanly. It will take two integer arguments, m and n. It checks to see whether m can be cleanly divided by n. It will return True if so and False otherwise. Now that we have the function signature, we can begin implementing the function. First, its definition:

```
1  def divides_cleanly(m, n):
```

Our first order of business is to make sure that n is not 0. If so, we could end up trying to divide by zero later in the function, which would cause the program to crash.

```
1  def divides_cleanly(m, n):
2      # Ensure we don't have division-by-zero issues.
3      if n == 0:
4          return False
```

If n is 0, then we will never be able to perform division, let alone to divide cleanly, so we return the value False. Now we need to check whether there

would be no remainder if we divided m by n (whether m % n == 0). If there is remainder, then the numbers do not divide cleanly; if there isn't remainder, they do divide cleanly.

```
1  def divides_cleanly(m, n):
2      # Ensure we don't have division-by-zero issues.
3      if n == 0:
4          return False
5      else:
6          if m % n == 0:
7              return True
8          else:
9              return False
```

If n is not 0, then the program proceeds to the else-statement on line 5. We then perform the boolean test in the nested if-statement on line 6, which checks whether there is no remainder. If this test is True, then the numbers divide cleanly and we return True (line 7); otherwise, the numbers cannot divide cleanly and we return False. This program involves three return statements, each handling a different set of circumstances.

Although this program works properly, there are several opportunities to simplify it. The first is to notice that the return on line 4 causes the function to immediately exit. If line 4 is executed, no code below that point in the function body ever executes. This means that we can actually eliminate the else-statement and the program will continue to run perfectly.

```
1  def divides_cleanly(m, n):
2      # Ensure we don't have division-by-zero issues.
3      if n == 0:
4          return False
5
6      if m % n == 0:
7          return True
8      else:
9          return False
```

Let's examine why we can safely eliminate the else-statement. There are two possible circumstances our program must handle: those in which n is 0 and those in which it is not.

- If n is not 0, the body of the if-statement on line 3 is skipped and the program continues to testing whether the numbers divide cleanly on line 6.

- If n is 0, the body of the if-statement on line 3 is executed, meaning that the program returns on line 4. Since a return statement causes execution to immediately return to the function's caller, the if-statement on line 6 never executes. This is a special property of return; if the return were

substituted for any other kind of statement (print, input, etc.), then the function would erroneously continue to the second if-statement.

We can actually simplify this function even further. Notice that the second if-statement has two branches, one of which returns the boolean value True and the other of which returns the boolean value False. We determine which of these branches to take based upon a boolean condition (m % n == 0). In other words, if a boolean expression is True, we return True; if it is False, we return False. Equivalently, we could just return the boolean expression itself.

```
1 def divides_cleanly(m, n):
2     # Ensure we don't have division-by-zero issues.
3     if n == 0:
4         return False
5
6     # If the numbers divide cleanly, return True.
7     # Otherwise, return False.
8     return m % n == 0
```

If the function gets to line 8, Python will evaluate the expression to the right of the return keyword. It will then return whatever value the expression evaluates to. Line 8 succinctly captures the logic we just described. When the expression to the right of return is True, then the numbers divide cleanly (meaning we should return True; when it is False, the numbers do not divide cleanly and we should return False. Either way, we return the value of the expression.

This trick dramatically simplifies the function, and it proves to be very handy in practice. Whenever you find yourself testing a boolean condition and return True if the condition is True and returning False if the condition is False, just return the condition. If you find the opposite case (if the condition is True return False and vice versa), return the boolean condition passed to the not function.

11.1.7 Scoping

Each function is a world unto itself. Any variables that you create within a function exist only within that function. They are destroyed at the end of each function call, and they cannot be accessed outside the function. The technical term for this behavior is scoping. Variables created within the scope of a particular function body exist only within that scope. The only way to transfer information between a function and its caller are through the narrow channels afforded by arguments and return values.

Why does Python make this restriction?

- In general, functions are designed to be as independent from one another as possible. If variables could be shared between the function body and the larger program, then functions could become so entangled with particular programs that they would be impossible to reuse elsewhere. The print

and len functions are completely self-contained, which is a large part of the reason we are able to use them as often as we do.

- It would be dangerous if an outside program could accidentally or maliciously modify the variables internal to a function. Imagine if the print function had an internal variable called s. Without scoping, your program could inadvertently create and modify variable called s that unintentionally breaks print's internal logic. Scoping ensures that programs and functions have separate namespaces—your program and the functions it calls can safely use the same variable names without worrying about overwriting each other's variables.

Python has several scoping rules that govern the way variables can be accessed from different parts of your program. In the sections that follow, we list those rules.

Rule 1: Variables created inside functions exist only inside functions. We have discussed this rule at length over the preceding sections. Any variable created in the function body, whether as an argument or through a variable declaration with the = operator, exists only for the life of the function call. These variables are not accessible from the larger program or from other function calls.

Rule 2: Every function call starts fresh. When a function is called, it begins with no internal variables. Any variables that it needs must be created anew every time the function is called. These variables can be created in the function definition to receive arguments or in the function body as normal declarations with the = operator. Either way, these variables are destroyed at the end of each function call and are freshly created the next time the function is called. In other words, variables inside a function do not save their values between function calls.

Rule 3: Variables from the larger program can be read (but not written) inside the function. This is where things start to get complicated. If a variable exists in the larger program, the function can access the variable's value but cannot modify it.

Consider the program below in which we write a function that calculates the day of the week for any date in 2017.

```
1 days_of_week = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
2 days_by_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
3 start_day = 0 # 2017 starts on Sunday.
4
5 # Calculate the day of the week for any date in 2017.
6 # month is a number between 1 and 12.
7 # day is a number between 1 and 31.
```

```
 8  def day_of_week(month, day):
 9      # Make month line up with days_by_month.
10      month = month - 1
11
12      # Determine how many days into the year we are.
13      days_into_year = 0
14
15      # Add all the days of the preceding months.
16      i = 0
17      while i < month:
18          days_into_year += days_by_month[i]
19          i += 1
20
21      # Add the days we are into this month.
22      days_into_year += day
23
24      # Add the starting day of the year.
25      days_into_year += start_day
26
27      # Subtract one day since January 1st is day 0.
28      days_into_year -= 1
29
30      # Calculate the eventual day of the year.
31      day = days_into_year % 7
32      return days_of_week[day]
33
34  # Tests.
35  print(day_of_week(2, 14)) # Valentine's Day.
36  print(day_of_week(7, 4)) # July 4th.
37  print(day_of_week(12, 25)) # Christmas.
```

Lines 1 through 3 define a few important facts about the calendar, listing the days of the week, the number of days in each month (2017 is not a leap year so February has 28 days), and the day of the week on which 2017 starts (the value 0 corresponds to 'Sun' in days_of_week).

The function itself begins on line 8. It is called day_of_week and takes two arguments, month and day. If we wanted to calculate the day of the week corresponding to Halloween, we would write day_of_week(10, 31). Line 10 converts months from the 1-12 range that we use to the 0-11 range that the list days_by_month uses.

We then calculate how many days month/day is into the year. For example, Valentine's day (2/14) is the 35th day of the year. We do so by adding up the number of days in the months preceding month (lines 16-20) and adding the number of days into the current month (line 22). We perform two corrections, subtracting 1 on line 28 (since January 1st is technically day 0) and adding the day of the week on which the year starts (line 25). Finally, we take the

remainder after dividing by 7 (the number of days in a week) to determine the eventual result.

Notice that Python lets us seamlessly refer to the variables days_of_week, days_by_month, and start_day inside the function. These variables are declared on the top-level of the program outside of any function. Variables declared in this way are said to be created in global scope, meaning they are part of the larger program. Python creates a smaller, nested scope for the body of each function. This nested scope has access to all of the variables declared in the containing global scope, like days_of_week, days_by_month, and start_day.

Incidentally, this rule explains how we can call a function from another function. At the beginning of this chapter, we discussed a program with two functions, print_author and print_quote.

```
1  def print_author():
2      print('-- Louis Sullivan')
3
4  def print_quote():
5      print('Form follows function')
6      print_author()
```

The print_quote function calls the print_author function. If each function were truly in its own closed universe, then the variable print_author would not exist within the body of print_quote. Since everything created in the global scope is readable within the scope of print_author's body (also known as its local scope), print_author can access the variable print_quote, which was created in global scope.

Rule 4: Variables from the larger program can be written using the global keyword. Rule 3 specifies only that variables created in the larger program can be read within functions. What happens if we try to write to them? Suppose we added some more code onto the end of the day-of-the-week program from the previous rule. Specifically, we want to write a function that will advance the calendar from 2017 to 2018. 2018 begins on a Monday. To capture this change, we would need to change start_day from 0 (Sunday) to 1 (Monday).

```
39 # Failed attempt to advance the calendar to 2018.
40 def broken_advance_to_2018():
41     start_day = 1 # 2018 begins on a Monday.
42     print('start_day in function: ' + days_of_week[start_day])
43
44 print('start_day before function: ' + days_of_week[start_day])
45 broken_advance_to_2018()
46 print('start_day after function: ' + days_of_week[start_day])
```

The output created by this addition to the program appears below:

```
start_day before function: Sun
```

```
start_day in function: Mon
start_day after function: Sun
```

Before calling the function, the first day of the year is Sunday as expected (first line of output). The function seems to change the value of start_day (second line of output), but this change doesn't appear to stick outside the function (third line of output). What happened? We have now encountered one of the strange idiosyncrasies of Python's scoping rules. Functions are free to read variables from the larger program. But when they try to write these variables, Python actually creates a new variable with the same name that exists only within the function scope. Line 41 actually creates a new variable that happens to also be called start_day but only exists within broken_advance_to_2018. It is completely separate from the variable called start_day that is created on line 3.

What if we wanted to modify the global version of start_day? Python provides a special command that allows your function full access to a global variable. Before trying to write to the variable start_day, we type the command

```
global start_day
```

which tells Python to use the version of start_day from the larger program rather than to create a new variable of the same name.

```
39 # Advance the calendar to 2018.
40 def advance_to_2018():
41     global start_day
42     start_day = 1 # 2018 begins on a Monday.
43     print('start_day in function: ' + days_of_week[start_day])
44
45 print('start_day before function: ' + days_of_week[start_day])
46 advance_to_2018()
47 print('start_day after function: ' + days_of_week[start_day])
```

This slightly-modified version of the previous function includes the use of the global keyword on line 41. The function has also been renamed to reflect the fact that it now works. The output from running this updated code is below.

```
start_day before function: Sun
start_day in function: Mon
start_day after function: Mon
```

Using the global keyword, we were able to modify start_day successfully.

A word of warning about the global keyword: it is a tempting convenience, but it violates one of the fundamental purposes of functions. As we discussed at the beginning of this section, functions are intended to be independent, self-contained units. As we stated earlier and repeat now, *the only way to transfer information between a function and its caller are through the narrow channels afforded by arguments and return values.* Sharing variables between functions and the global scope can occasionally be useful, but more often than not it creates undesirable entanglements between ostensibly independent functions and

the programs they inhabit. We strongly discourage use of the global keyword as bad style, although we acknowledge that it is important to know that it exists and that there are rare occasions where it is useful.

Rule 5: You can end up with multiple variables that have the same name. Suppose that your program has a variable named x in the global scope. You now try to define a function with an argument called x. What happens? Your program will actually have two separate variables, each of which is called x. In the global scope, the name x will refer to the variable you defined before the function was created. Inside the function, x will refer to the argument. The argument's name will shadow the other x, rendering it inaccessible until after the function is done executing.

Let's consider a concrete realization of this example.

```
1  x = 2
2
3  def increment(x):
4      return x + 1
5
6  print(increment(5))
```

On line 1, we define a variable called x as promised; x is assigned the integer value 2. On line 3, we define a very simple function called increment; it takes a single integer argument and returns the value of that argument plus 1. We have chosen to call the argument x. Finally, on line 6, we try calling the function. It neither crashes nor prints an error complaining about the existence of two different variables; it simply successfully prints the output 6.

To get at the heart of shadowing, suppose we added one more line to the program.

```
7  print(increment(x))
```

This line of code calls increment with the argument x. Which x does this refer to? Since we are in the larger program outside of any function, the name x refers to the variable created on line 1 (the name x from within the function only exists within the function scope, so it does not exist here). When Python executes the function body, it jumps to line 3 and creates another variable called x. This variable is separate from the global x. Since we passed the global x as the argument to the function call, its value is copied to the local x that is accessible within the function scope.

Shadowing may seem complicated, but it is a valuable measure that protects the independence of functions. When designing a function, you do not need to worry about accidentally creating variables that conflict with names in the larger program.

11.2 In Praise of Functions

Throughout this chapter, we have foreshadowed some of the many benefits that functions offer. At this juncture, you finally have enough knowledge to fully appreciate the vital role functions play in building large programs.

Code reuse. From a utilitarian perspective, functions make it possible for you to write a block of code once and reuse it as often as desired. By this (admittedly still early) point in your programming career, you have already encountered many situations in which you had to repeat the same few lines of code over and over again with slightly different variables or subexpressions. Until now, you had to resort to copying and pasting a block of code throughout your program with minor alterations. Now that you have functions, however, you can write a function that captures the repeated behavior and call it each time you want to reuse the code.

In fact, it is considered bad style to do anything but write a function in these circumstances. Copying and pasting code is dangerous operation that tends to lead to subtle bugs in practice. Imagine the following scenario: you write a few lines of code to perform a tricky operation and copy it several times throughout your program. A few days later, you need to tweak this operation slightly; perhaps you found a bug or have to handle a previously-unanticipated scenario. You will now need to find every location where you copied the code and modify each one in exactly the same way. Professional programmers will admit that two extremely common bugs tend to result:

1. When making the same change in several places, chances are you will make a mistake in at least one.

2. Finding every place where the code occurs is very difficult. In practice, you're likely to miss one or two places.

If you were to capture this repeated code in a function instead, you would only need to make the change once: inside the function body. Not only is doing so much safer, but it is also far easier. Programmers would say that capturing repeated code in functions makes programs more maintainable—easier to modify or fix down the road. Since it is easier to tweak and upgrade, maintainable code tends to have a longer lifetime, allowing you to reap greater rewards from the time you initially invested writing the program.

So far, we have only mentioned reusing code within a single program. Functions offer an even more powerful capability: reusing code across programs. If you write a particularly useful function, you can call it from any program you write going forward. We will discuss the precise mechanics for doing so in the last section of this chapter. From the very first pages of this book, we have relied on functions written in this way: print, len, input, etc. Imagine if you had to build your own len function every time you wanted to use it; even worse, imagine if you didn't have functions at all and had to copy and paste the code underlying len.

If you write a function that you find as helpful as print or len, you are able to reuse it across programs. You can also draw on the thousands of programmers who have written useful functions themselves. Later in this book, we will discuss downloading modules that Python programmers have written and published for manipulating PDF documents or interacting with websites. All of this is possible thanks to functions.

Structure. From a design perspective, functions provide a natural tool for breaking your programs down into smaller pieces. Writing a big program can quickly get complicated. A tangle of variables, datastructures, and transformations can quickly become too much to keep track of. The human brain simply isn't capable of remembering every detail of thousands of lines of code at once. Big programs tend to become messy, bugs become common and hard to find, and the entire edifice eventually collapses upon itself. In short, big, monolithic programs are unmaintainable.

Yet big programs are common in practice. The Linux operating system consists of tens of millions of lines of code; Microsoft Windows is estimated to be over a hundred million lines. How do these professional engineers avoid the pitfalls we have just described? They break big problems down into small pieces. Microsoft Word is not a single large program. It is a series of smaller programs that format text, check spelling, add footnotes, save, and print. By decomposing a large program down into smaller, manageable pieces. Each of these components aims to accomplish one specific goal.

Functions are purpose-built to make this design philosophy possible. Each element of a large program can be broken down into a sequence of smaller operations, each of which is captured in a function. In the same fashion, those functions can be broken down further. Eventually, these pieces will be small enough to implement easily in a handful of lines of code. This top-down design approach—starting with larger goals that are subsequently subdivided into smaller ones—is very effective in practice. It makes it possible to take programs of daunting scale (like the program that Google uses to make searching the entire web possible) down into tractable pieces (a program that downloads a webpage and looks for important keywords).

Abstraction. One way to look at program structure is through layers of abstraction. It would be very difficult for every programmer working on Microsoft Word (thousands in total) to have to think about the intricacies of spelling and grammar in 50 different languages. Instead, someone has written a check_grammar function that handles these concerns, giving other programmers the freedom to write their code under the assumption check_grammar works properly. The same is true for the save_document and print_document subsystems. Other programmers can build on these functions further, creating even higher levels of abstraction.

The key idea behind abstraction is that it is often better not to know the details. Instead it is preferable to write a series of functions that manage the

low-level details and build on these functions, creating layers that accomplish bigger and bigger goals. For example, you might write a set of functions that manage the details of formatting dates, times, and money properly for printing. You could then write a set of functions that calculate and print bills using these functions. In turn, an even larger multi-client billing program could rely on those functions. One could even imagine writing a legal office-management program that uses the billing subsystem. Each function need only be concerned with the layers immediately above and below it.

Consider a concrete example. If we have used the print function so much since the start of this book, why have we never bothered to look at the underlying code that it uses to do its job? Simply put, this code isn't important. As long as the function does its job, it isn't worth our time to worry about how it manages to do so. For all we know, the print function may not even be written in Python.[1] Depending on the version of Python you have and the operating system you are using, your computer's implementation of print could be completely different. As long as the function always works, however, these differences are inconsequential. If the designers of Python came up with a brilliant new way to reimplement the code underlying the print function, they could swap in the new version tomorrow and you would never notice the difference.

In summary, to use a function properly, all you need to know is its signature—its name, the arguments it expects, the effects it has, and the value it returns. The lesson of abstraction is that, not only is this information necessary, but it is also sufficient. Knowing anything more about a function is usually a waste of time, energy, and brain capacity better spent elsewhere.

Abstraction is also what makes it possible to write programs as a team. Typically, a large project is broken down into smaller functions. Once the signatures of those functions are specified, each member of the team can independently write her own functions, confident that, so long as everyone else's functions adhere to their signatures, the individual components can be put back together into a complete working product.

Isolation. Each function is a self-contained unit of computation. As we have discussed in-depth, Python takes strong measures to ensure that functions are isolated from the outside world and makes it extraordinarily inconvenient to violate these boundaries. Doing so is part of what makes it possible to reuse functions within or across programs—by ensuring functions are independent entities, they can be used safely in a wide range of circumstances.

This isolation also serves a second, equally-valuable purpose: it makes testing and debugging programs dramatically easier. Which would you rather examine when searching for a bug: a multi-hundred line tangle of code or a function with a dozen lines? If you build a program out of smaller functions, it's easier to trace bugs to particular parts of your code and, afterward, to pinpoint the exact causes. A function containing a short block of code intended to accomplish

[1] Many core Python functions are actually written in a language called C. The Python interpreter, which executes your Python programs, is also written in C.

a single, precise purpose is far easier to maintain. When testing this sort of function, you can be sure that you have tested every possible input. When debugging, there are only a handful of places a bug can hide. Small functions make it possible to achieve correctness—to be nearly certain that a function works perfectly. A large program constructed from these building blocks will itself be very reliable.

11.3 Advanced Features of Functions

You now have all of the tools you need to enjoy the benefits of functions. Although functions may already seem overly feature-laden, Python offers a few more conveniences that make functions even more powerful. Here, we discuss several that you are likely to see often in practice.

11.3.1 Multiple Return Values

There is one fundamental limitation of return values as we have presented them so far: each function can only return one value. Suppose we wanted to write a function that calculates some basic descriptive statistics about a list of numbers: the mean (the average), the median (the middle number), and the range (the difference between the biggest and smallest numbers). We would need a way to return three values rather than just one. Our first attempt could be to combine these three values into a list or dictionary, thereby making them into a single value that we can return.

```
1  def stats(numbers):
2      # Calculate the mean.
3      mean = sum(numbers) / float(len(numbers))
4
5      # Calculate the median.
6      # First, sort the numbers.
7      sorted_numbers = sorted(numbers)
8
9      # If we have an odd length list, pick the middle one.
10     if len(numbers) % 2 == 0:
11         median = sorted_numbers[len(numbers) / 2]
12
13     # If we have an even length list, average the middle two.
14     else:
15         middle1 = sorted_numbers[len(numbers) // 2]
16         middle2 = sorted_numbers[len(numbers) // 2 - 1]
17         median = (middle1 + middle2) / 2.0
18
19     # Calculate the range (biggest minus smallest).
20     range = sorted_numbers[-1] - sorted_numbers[0]
21
```

```
22      return {'mean':mean, 'median':median, 'range':range}
23
24  # Calculate the statistics for a particular list of numbers.
25  numbers = [2, 7, 9, 18, 28, 1, 82, 8, 4, 5, 9]
26  outcome = stats(numbers)
27  mean = outcome['mean']
28  median = outcome['median']
29  range = outcome['range']
```

The function involves a small amount of math. On line 3, it calculates the average of the numbers by adding them together and dividing by the length of the list. On lines 8 through 16, it calculates the median—the middle number. It does so by sorting the numbers (line 7) and then picking the middle number (line 11). If there are an even number of numbers, then it averages the middle two numbers (lines 15-17). Finally, it calculates the range by subtracting the smallest number from the biggest (line 20). It returns all three numbers in a dictionary (line 22).

This approach does work—lines 26 to 29 will manage to extract the function's return values—but it is unwieldy. Python offers a better way. When we have multiple return values, we can separate them with commas after the return keyword. When retrieving these return values, put multiple variables to the left of the = operator. The return values and the variables that retrieve them need to be in the same order.

```
22      return mean, median, range
23
24  # Calculate the statistics for a particular list of numbers.
25  numbers = [2, 7, 9, 18, 28, 1, 82, 8, 4, 5, 9]
26  mean, median, range = stats(numbers)
```

On the updated line 22, we place the three values we want to return after the return keyword separated by commas. On line 26, we capture those three returned values by placing three variables to the left of the = operator. Although we gave the variables the same names inside the function and in the program as a whole, we are under no obligation to do so. The code below would also work.

```
26  the_mean, the_median, the_range = stats(numbers)
```

The number of variables that you place to the left of the = operator must be exactly the same as the number of values that the function returns. If you were to only place two variables

```
26  mean, median = stats(numbers)
```

you would see the following error:

```
Traceback (most recent call last):
  File "stats.py", line 26, in <module>
    mean, median = stats(numbers)
ValueError: too many values to unpack (expected 2)
```

Python complains that it has three values to "unpack" (the three values returned by the function) but only two variables in which to do so. You would see a similar error if you provided too many variables.

Sometimes, you will only care about some of the values that a function returns. For example, what if you were only interested in the mean and range of a list of numbers? In place of the unwanted variable, you can leave the _ (underscore) character, which tells Python to throw away the value.

```
26 mean, _, range = stats(numbers)
```

When you use multiple return values, Python is actually turning the comma-separated list of return values into a tuple under the hood. You will recall that a tuple is identical to a list, except that it is immutable (it cannot be modified). Python then unpacks this tuple into the variables you place to the left of the = operator.

11.3.2 Keyword Arguments

How does Python know which arguments from a function call are assigned to which variables in the function itself? So far, it is through order. The first argument is assigned to the first variable, the second argument to the second variable, etc. As you begin to learn more and more functions, it can get hard to keep track of the order of variables in addition to their types and purposes. Python provides an alternate way to pass arguments to functions that alleviates the need to remember their order. Instead, you can use their names.

Imagine that we have further embellished the greeting function from earlier in this chapter.

```
1 def greet(greeting, title, name, ending):
2     print(greeting + ', ' + title + ' ' + name + ending)
```

In addition to specifying a name and greeting, the program now asks for the person's title ('Mr.', 'Professor', 'Dr.', etc.) and the symbol with which to end the greeting ('.', '!', etc.). At this point, the function has four arguments, and memorizing the exact order in which they appear is tricky.

As long as we can remember the names of the arguments as they are written on line 1, however, we can still call the function.

```
4 greet(greeting='Good evening', title='Mr.', name='Bond', ending='...')
```

In the function call, in addition to providing the values of the arguments, we also specify the variables they correspond to in the function definition on line 1. For each argument, we write the name of the variable, the = symbol, and the name of the argument; we separate these statements with commas. Python interprets this comma-separated series of variables and arguments as the assignments it should implicitly perform before the function body is executed. This style of passing arguments is known as using keyword arguments. As expected, the program would print

```
Good evening, Mr. Bond...
```

Since we have now specified the variables that each argument corresponds to, order no longer matters. The following function calls will all produce the same result.

```
greet(greeting='Good evening', title='Mr.', name='Bond', ending='...')

greet( title ='Mr.', greeting='Good evening', name='Bond', ending='...')

greet(ending ='...',  title ='Mr.', name='Bond', greeting='Good evening')
```

The only requirements to use this style of function call are that all variables must be accounted for and that the variable names must exactly match those on the first line of the function declaration.

Python also permits you to mix passing arguments in order and passing arguments by name. For example, you could call greet as follows:

```
greet ('Good evening', 'Mr.', ending ='...',  name='Bond')
```

The first two arguments do not have corresponding variables. Python will automatically associate the first argument with the first variable (greeting) and the second argument with the second variable (title). As soon as you begin specifying keyword arguments (starting with ending in the example above), Python will make the associations you specify.

Once you start using keyword arguments in a function call, every argument from that point needs to be a keyword argument. If we typed the following function call

```
greet ('Good evening', 'Mr.', ending ='...',  'Bond')
```

Python would produce the error below:

```
  File "greeting.py", line 4
    greet ('Good evening', 'Mr.', ending ='...',  'Bond')
                                                      ^
SyntaxError: positional argument follows keyword argument
```

The error is self-explanatory—Python complains that, after the keyword argument ending, we supply an argument without a keyword (a positional argument).

11.3.3 Default Arguments

Often, you will write a function where one of the arguments almost always takes on the same value. Python provides a way to give those arguments default values so that a caller can simply ignore those arguments in all but the rarest of situations. For example, the print function actually takes five arguments. We have only ever used the first argument—the string to be printed. In addition to three arguments that control esoteric technical details, it takes an argument called end that controls the symbol that is printed after the string. By default, end is set to '\n', meaning that each call to print ends with a newline. We could

change that value to '!!!\n' to end a print statement with three exclamation points or '' (the empty string) to eliminate the new line entirely.

To override the default value of an argument when calling a function, simply set it using a keyword argument. For example, to call print with a different ending, we could write the following:

```
print('hello, world', end='!!!\n')
```

This statement overrides the default value of the argument called end. It will print the following:

```
hello, world!!!
```

To print several strings on the same line, we could change the value of end to the empty string.

```
print('This ', end='')
print('is ', end='')
print('on ', end='')
print('one ', end='')
print('line.')
print('This is on the next line.')
```

The first four function calls override Python's default behavior to create a new line after each print statement, meaning the words appear on the same line. Only after the fifth print statement, where we do not override the default value of end, do we create a new line.

```
This is on one line
This is on the next line.
```

Numerous functions in the Python standard library, including many that we have encountered, have a variety of arguments with helpfully-chosen default values. By looking through the Python documentation[2] you can explore and manipulate these arguments. Default arguments make it possible to provide a range of choices without overwhelming users with too many arguments to keep track of.

You can easily implement default arguments in your own functions. Default values are specified in the function's definition. Default arguments are written in the same way as keyword arguments, except that default arguments appear in the function definition and keyword arguments in a function call. When writing a variable's name, follow it with the = operator and the default value. To make this concrete, we return to greet function from the previous section.

```
1 def greet(greeting, title, name, ending='.'):
2     print(greeting + ', ' + title + ' ' + name + ending)
```

The first three arguments do not have default values; a caller will have to provide a value for each of them. The fourth argument, ending, has a default value, '.'.

[2] `https://docs.python.org/3/`

If we only provide the first three arguments, the function will still run properly under the assumption that ending should have the value '.'. The function call

```
greet('Good evening', 'Mr.', 'Bond')
```

is perfectly acceptable to Python and will print

```
Good evening, Mr. Bond.
```

We could set a different default value and get a different result from the same function call. For example, if we modified the function definition to

```
1 def greet(greeting, title, name, ending=' :)'):
2     print(greeting + ', ' + title + ' ' + name + ending)
```

then the same function call would produce the output

```
Good evening, Mr. Bond :)
```

Just as with keyword arguments, once you start specifying default arguments, every argument that follows needs a default value. If we tried to write the following function definition with a default value for title in addition to ending

```
1 def greet(greeting, title='Mr.', name, ending=' :)'):
2     print(greeting + ', ' + title + ' ' + name + ending)
```

Python will complain as soon as it tries to create the function:

```
  File "greeting.py", line 1
    def greet(greeting, title='Mr.', name, ending):
                                    ^
SyntaxError: non-default argument follows default argument
```

This error is almost identical to the one we encountered with keyword arguments. Python complains that an argument lacking a default value (namely, name) followed an argument with a default value (title).

11.3.4 Recursion

Python functions can call themselves. This process, known as recursion, is far more than just an interesting footnote—it is an entirely different way of approaching many problems we previously solved with loops. As it turns out, recursion is even more powerful than loops, meaning that a function that calls itself can express certain calculations that are difficult or inelegant to write otherwise. Recursive functions can sometimes look mind-bending, but they are able to capture complex computation with remarkable brevity.

Consider a toy program similar to that with which we introduced while-loops in Chapter 7. It aims to print the numbers 1 through 5.

```
1 x = 1
2 while x <= 5:
3     print(x)
4     x = x + 1
```

Now consider a recursive function that accomplishes the same task.

```
1  def print_numbers(x):
2      # First, print all numbers smaller than x.
3      # Stop calling self when x reaches 1.
4      if x > 1:
5          print_numbers(x − 1)
6
7      # Finally, print x.
8      print(x)
9
10 print_numbers(5)
```

The function print_numbers takes a single argument, x, which tells the function to print the numbers 1 through x in order. Inside the function itself, we first test whether x is greater than 1 before making a recursive call. This test ensures that the recursion eventually stops—that when x reaches 1 the function stops calling itself and exits. It serves the same purpose as the boolean test in the while-loop, ensuring that the function eventually finishes so the program can continue.

If x is indeed greater than 1, the function calls itself on line 5 with the argument x − 1, which will print the numbers 1 through x − 1. Once those numbers are printed, the function's only remaining responsibility is to print the number x (line 8) and exit.

On line 10, the program calls print_numbers for the first time with the argument 5. The function receives this argument on line 1 and successfully tests the if-statement's condition on line 4. Since 5 is greater than 1, the function continues to line 5, where it calls itself on the argument 4, which should print the numbers 1 through 4. Since a new function call has started, execution returns to line 1 with the new argument.

This process repeats for each successive argument down to 1. When print_numbers is called on 1, the if-statement's condition fails and the program proceeds to line 8, where it prints 1. Once this function call is done, execution returns to the location that called it—line 4 of the function call where x was 2. Python proceeds to line 8, where it prints 2. With this function call done, execution again returns to the location that called it—line 4 of the function call where x was 3. The number 3 is printed, and the process continues unwinding until 5 is printed, at which point all function calls have been completed and the program ends.

Each execution of the function is equivalent to one iteration of the loop. Likewise, every time the if-statement test succeeds and the function calls itself is identical to a while-loop test that passes, leading to another loop iteration.

There are a few common recursion mistakes that we could make with this program. The first is to print the number *before* making the recursive call.

```
1  def print_numbers(x):
2      # First, print x.
```

```
3       print(x)
4
5       # Finally, print all numbers smaller than x.
6       # Stop calling self when x reaches 1.
7       if x > 1:
8           print_numbers(x − 1)
9
10  print_numbers(5)
```

This function would print the numbers backwards.

```
5
4
3
2
1
```

On each call to the function, x is printed on line 3 before the recursive call is made on line 8. This means that, when x = 5, the number 5 will print before the program ever tries to print the numbers 1 through 4 via the recursive call. Order matters within recursive functions. Code that is written before the recursive call will occur top-down (printing the number 5 before the number 4), while code that is written after the recursive call will occur bottom-up (printing the number 4 before the number 5). When using recursion, it is critical that you keep track of where in the function code is placed.

Another common mistake is to forget the if-statement entirely.

```
1   def print_numbers(x):
2       # First, print all numbers smaller than x.
3       print_numbers(x − 1)
4
5       # Finally, print x.
6       print(x)
7
8   print_numbers(5)
```

Here is the output Python will produce when this program is run.

```
...
  File "test_recursion.py", line 3, in print_numbers
    print_numbers(x − 1)
  File "test_recursion.py", line 3, in print_numbers
    print_numbers(x − 1)
  File "test_recursion.py", line 3, in print_numbers
    print_numbers(x − 1)
RecursionError: maximum recursion depth exceeded
```

Since we neglected to give the program any mechanism to stop, the program kept calling itself forever. Unlike loops, Python places a limit on the number of nested

function calls a program can make at once, after which it forces the program to crash. (In many languages, this error is known as a stack overflow.) That is precisely what happened here—print_numbers called itself so many times that it reached Python's "maximum recursion depth". This error is exactly like an infinite loop—if we had told a loop to continue while True, it would never finish. Just like loops, all recursive functions need an exit condition that will eventually be triggered.

As we mentioned before, recursive functions are actually more powerful than loops—you can write a wider range of programs with recursion than with loops. Each iteration of a loop proceeds in a linear fashion, one after the other, forming a chain from start to finish. A recursive function, on the other hand, can call itself multiple times, meaning that one iteration can lead to two, three, or many additional iterations, each of which can create many more themselves. Recursive functions tend to branch outwards as the number of recursive calls multiply; loops have no equivalent behavior.

Consider the problem of calculating whether a number is a valid American football score. For example, is it possible for a football team to score 19 points? For the sake of simplicity, assume that the only ways of scoring in football are through touchdowns (worth 7 points) and field goals (worth 3 points). To figure out whether 19 is a valid football score, we should ask how a team could have arrived at that score. If the last points the team scored were via a touchdown, then we know that the team previously had 12 points. Alternatively, if the last points the team scored were via a field goal, then the team previously had 16 points. At this juncture, our program should ask whether 12 and 16 are valid football scores. Repeating the same method, the program would then need to ask whether 5, 9, 9, and 13 are valid football scores (subtracting a field goal and a touchdown each from 12 and 16). This branching, where each follow-up question results in two more, is difficult or impossible to expression in a loop, but we can easily do it with recursion.

```
1  def is_football_score(points):
2      # Zero points is a valid football score.
3      if points == 0:
4          return True
5
6      # Less than zero points is not a valid football score.
7      elif points < 0:
8          return False
9
10     # If there are more than zero points, try subtracting a
11     # field goal or a touchdown.
12     else:
13         fieldgoal = is_football_score(points - 3)
14         touchdown = is_football_score(points - 7)
15
16         # If either of these options yields a valid score,
```

```
17      # return True. Otherwise, return False.
18      return fieldgoal or touchdown
```

The function receives the number of points as its argument. It returns True if the score is, indeed, a valid football score and False otherwise. Lines 2 through 8 ensure that the function eventually stops calling itself. If the number of points is exactly zero, then it must be a valid football score; if it is less than zero, then it is not.

Finally, lines 10 to 19 handle the final case—when the number of points is greater than zero. On lines 13 and 14, the function recursively checks whether subtracting a touchdown or a field goal results in a valid football score. These calls capture the step of asking whether 12 and 16 are valid football scores when points is 19. If either of the recursive calls comes back True, then the boolean or on line 19 will also evaluate to True, making that the function's return value. Otherwise, the function will return False.

This function elegantly captures a complex computation in only a few lines of code. Furthermore, it is easy to understand, and future readers will have little trouble making sense of how the code works. When structured properly, recursion tends to produce similar results. Recursion isn't an essential for programming in the same fashion as variables or if-statements, but it offers an alternate way to solve many otherwise messy problems. Even if you choose not use recursion in your everyday programming, it is important to know that it exists and how to read recursive functions.

11.4 Modules

Throughout this chapter, we have extolled the virtues of writing functions that can be reused across multiple programs. In this section, we give you the machinery to do so. We can package functions into bundles called modules which can then be imported by other Python programs.

11.4.1 Creating modules.

A module is simply a Python file consisting entirely of functions. For example, the program below is a Python module with a few basic statistical functions borrowed from earlier in this chapter.

```
1  # A module of statistical functions.
2
3  # Calculate the mean of a list of numbers.
4  def mean(numbers):
5      return sum(numbers) / float(len(numbers))
6
7  # Calculate the median of a list of numbers.
8  def median(numbers):
9      # First, sort the numbers.
```

```
10      sorted_numbers = sorted(numbers)
11
12      # If we have an odd length list, pick the middle one.
13      if len(numbers) % 2 == 0:
14          return sorted_numbers[len(numbers) / 2]
15
16      # If we have an even length list, average the middle two.
17      else:
18          middle1 = sorted_numbers[len(numbers) // 2]
19          middle2 = sorted_numbers[len(numbers) // 2 - 1]
20          return (middle1 + middle2) / 2.0
21
22  # Calculate the range (biggest minus smallest).
23  def range(numbers):
24      return max(numbers) - min(numbers)
```

Notice that, on its own, this Python file doesn't do anything. If we were to run it, it would produce no output. It simply defines three functions, none of which are ever used. This program is an exemplary module. It does not exist for its own purpose; rather, it defines three functions with the expectation that other programs will import and call them. For now, let's save this code to a file called stats.py.

11.4.2 Using modules.

Now that our module is complete, we need to start a new Python file to make use of the functions. Before we can use the functions from the module inside our new program, we need to take care of a couple of formalities:

1. The module (stats.py) must be in the same directory as the Python file we're composing. Otherwise, Python won't be able to find it.

2. We need to explicitly tell Python that we want to use the functions from stats.py by importing it. To do so, we write the import keyword followed by the module's filename (without the .py part) at the top of our new program. In other words, the first line of our new program is:

```
1  import stats
```

   The import statement does not technically need to be at the top of the program; it only needs to appear before functions from the module are called. However, it is common practice across many programming languages (including Python) to put all import statements at the very beginning.

   We have used import statements in a few places throughout the preceding chapters. For example, we needed to type import sys in order to access command-line arguments. This section should finally explain what Python was doing behind the scenes.

To call a function from a module we need to precede the function's name with the name of the module and a dot. For example, the following program uses our stats module to print the average of a list of numbers:

```
1 import stats
2
3 numbers = [2, 7, 9, 18, 28, 1, 82, 8, 4, 5, 9]
4 average = stats.mean(numbers)
5 print(average)
```

It can sometimes get tedious to type the name of the module every time we wish to use one of its functions. Although typing the word stats is a small inconvenience, a less generous programmer might have named the module

```
helpful_statistical_functions
```

You can rename the module when you import it by using the as keyword:

```
1 import helpful_statistical_functions as stats
```

If the module were stored in a file called helpful_statistical_functions.py, the preceding command imports the module and then renames it to stats, meaning we can refer to the stats.range function rather than the

```
helpful_statistical_functions.range
```

function.

Python makes it possible to dispense with the module name entirely when calling an imported function. You can import a specific function from a module with the from keyword.

```
1 from stats import range
```

This statement imports only the range function; median and mean would have to be imported separately. Importing in this fashion eliminates the need to write stats.range; you can simply use the name range to refer to the so-named function from the stats module. You should use this style of import sparingly and with caution, however. Although typing the module name is cumbersome, it protects you from accidentally overwriting an existing function that happens to share the same name. For example, the preceding import statement just overwrote Python's built-in range function, which generates a list of numbers. Had we used a normal import statement, the built-in function range would have remained separate from the module's stats.range function.

A warning. One word of caution about the import statement: it causes all code in the module to immediately execute. If the module contains top-level print statements, they will execute as well. A properly-designed module should contain only function and variable definitions; when executed on its own, it should not produce any perceptible effects (like calls to input or print).

11.5 Conclusion

We have now reached the end of our lengthy journey through functions and modules. It is difficult to understate the value of functions in programming. As we have repeated time and again throughout this chapter, functions make it possible to write any program that stretches beyond a few dozen lines of code.

Functions shatter many of our preexisting rules about the way Python executes programs. We spent many pages exhaustively exploring the many complicated rules that govern functions, their arguments, their return values, and the way they fit into Python programs. We even gave you the ability to write your own modules of Python functions that you can share between multiple programs. Until this chapter, code was frozen permanently in each program, invokable only fro the command line. Functions make code into a malleable substance that can be created, modified, distributed, and called from within other code.

In the process of doing so, we explained many of the remaining mysteries about Python inner workings. How do functions work internally? What happens when I call a function? Why are there so many different styles of passing arguments? How does print or len work under the hood? How is the Python standard library written?

11.6 Cheatsheet

Terminology

Function—A bundle of Python code somewhere else on your computer that you can call upon to help your program perform some task.

Name—The variable name used to refer to the function. For example, the print function's name is print. Function names can be copied and overwritten just like any other variable.

Call—To cause a function to execute. For example, writing print('Hello') is calling the print function. When a function is called, Python momentarily stops executing the program at the call site, transfers to the function's code, executes that code until it finishes, and then returns to the call site to continue executing normally. Functions can call other functions as their own helpers when necessary.

Argument—A value that is passed to a function to be used in the course of the function's computation. For example, writing print('Hello') pass the argument 'Hello' to the call to the print function. Functions receive one or more arguments separated by commas in the function call. Order matters—arguments must be passed in the correct order.

Return Value—The value to which a function call evalutes. For example, a call to the len function (i.e., len('hello')) returns an integer (i.e., it returns 5). Some functions, like print, do not have a return value; they instead evaluate to None.

Effect—A result of calling a function other than the value it evaluates to. Effects include printing, asking for input, changing the value of a reference-type argument, connecting to a website, modifying a file, etc. All functions either return a value, have an effect, or both; if a function has neither, it doesn't do anything.

Signature—Everything you need to know about a function to use it properly. Specifically, (1) its name, (2) the type and number of arguments it expects, (3) whether it returns a value and the type of the return value, (4) whether it modifies any of its reference-type arguments, and (5) any effects that it has.

Defining Functions

```
1 def <name>(<arg1>, <arg2>, <arg3>):
2     <function body>
```

```
3       <more function body>
```

A function is created in five steps:

1. The def keyword.
2. The name of the function (any valid variable name).
3. Parentheses containing comma-separated variable names corresponding to each argument the function expects.
4. A colon.
5. A code block of one or more indented lines containing the code that should execute whenever the function is called.

This code creates the function under the specified name. The function does not execute until it is called. When the function is called with arguments, the arguments provided at the call site are stored in the corresponding variables listed in the function definition's first line. Those variables are then available for the function body to use.

Return Values

A function returns a value using the return keyword.

```
1  def <name>(<arg1>, <arg2>, <arg3>):
2      <function body>
3      <more function body>
4      return <expression>
```

When Python encounters the return keyword, it immediately exits the function and causes the function call to evaluate to the expression that follows the return keyword. Anything after the return statement will be ignored. A function can have multiple return statements; often, an if statement will choose between two possible return values.

You can return multiple values by separating the expressions to be returned with commas. This will implicitly create and return a tuple. You can receive these elements by assigning the result of the function call to multiple, comma-separated variables to the left of the equals sign.

```
1  def stats(list_of_numbers):
2      # Compute some statistics.
3      return mean, median, mode
4
5  salary_mean, salary_median, salary_mode = stats(salaries)
```

Value and Reference Arguments

When argument is passed to a function, the value of the argument is moved to the newly-created variable as if it were a normal assignment. If the argument was a value-type, then the argument is copied, meaning that modifying the argument variable inside the function does not modify anything outside the function. If the argument was a reference-type, then the reference is copied and modifying the argument variable will modify the value that was passed as the argument.

Keyword Arguments

Normally, you indicate which arguments should be stored in which argument-variables by passing arguments in the same order as the argument-variables. In other words, the first argument is automatically stored in the first argument-variable.

You can explicitly pass a particular argument to a particular argument-variable by using the = operator in the function call. Suppose we have a function that prints greetings:

```
1 def greet(person, greeting):
2     print('{0}, {1}!'.format(greeting, person))
```

For example, the expression greet('Mr. President', 'Happy birthday') would print 'Happy birthday, Mr. President!'. Rather than passing the function arguments in order, you could write

```
greet(greeting='Happy birthday', person='Mr. President')
```

This style of function call explicitly states which arguments go to which argument-variables, so order no longer matters.

You can mix passing arguments by position and passing arguments by name, but once you start passing by name every argument must be passed by name.

Default Arguments

A function can specify default values for some of its argument-variables. If the arguments aren't provided, the corresponding argument-variables are set to the default values. To specify a default value, follow the name of the argument-variable in the function definition with the = operator and the default value.

```
1 def greet(person, greeting='Good morning'):
2     print('{0}, {1}!'.format(greeting, person))
```

When the function is called with just one argument, that argument will go to person and greeting will take on the value 'Good morning.'.

Default arguments superficially look like keyword arguments, but default arguments are written in the function definition while keyword arguments are used when calling a function.

Scoping

Scoping is the process of determining which variables are accessible at any given location or point of time in a program. We identify five scoping rules.

1. Variables created inside functions exist only inside functions. The function's arguments and any other variables created inside the function are destroyed as soon as a particular call is finished executing. They are created anew for each function call.
2. Every function call starts fresh. No internal variables (other than the arguments) are available, and the previous values of any variables are wiped out.
3. Variables from the larger programs can be read (but not written) inside the function. If a variable exists at a level bigger than the function's body, that variable can be used but not modified.
4. Variables from the larger program can be written using the global keyword. That is, to write to a variable x that exists in the larger program as described in Rule 3, first write the statement global x on its own line; then, you will be able to write to the variable x that exists in the large program. If you don't do this and try to assign to x, Python will create a new variable also named x that only exists inside your function.
5. You can end up with multiple variables that have the same name. When you assign to a variable for the first time inside a function, Python creates a new variable with its own storage area, regardless of whether that name exists elsewhere in the program.

Recursion

A function can call itself. This behavior can mimic a loop (when every function call calls itself once until some condition is met) or something more sophisticated (every function call calls itself more than once). Recursion is strictly more powerful than looping (in that it can express a wider range of programs).

Benefits of Functions

- Code reuse. A function can be used over and over again in a program or across programs, saving time and eliminating repetitive code and making it easier to upgrade code later.
- Structure. Functions break programs down into smaller, easier-to-manage, easier-to-validate pieces. It is generally considered gpod software design to build your programs out of these smaller building-blocks.
- Abstraction. Functions hide away unnecessary detail, making it easier to focus on the important aspects of programming and to later seamlessly upgrade these detailed parts of the function.

- Isolation. Functions are designed to be disentangled from any one program, meaning they can be built, tested, and validated independently.

Modules

A module is a set of Python functions that are available for use by other Python programs.

To create a module, create a Python file that only contains functions (no top-level printing, input, etc.). Suppose this module is saved to the file my_module .py. Another Python file in the same directory can use the functions in that module.

1. Import the module to make it available for use in Python by writing import my_module (where my_module is replaced with the name of the module's Python file with the .py part removed).

2. Functions in the module can be called by writing my_module.function_name (where my_module is replaced with the name of the module).

If the module has an inconvenient name, you can give it a new name within the program that uses the module by writing

import my_module as mm

Functions from the module can be referred to using the prefix to the right of the as operator (i.e., mm.my_function).

Keywords

Body—The indented code following a function definition that executes when the function is called.

Global scope—Anything that is defined at the very top-level of a Python program outside of any function.

Global variable—A variable defined in global scope.

Impure—A function is impure if it modifies one or more of its arguments or if it has effects.

Local scope—Anything that is defined within a scope smaller than global scope, for instance inside a function.

Local variable—A variable defined in a local scope.

Namespace—The set of variables that exist in a particular context. Functions have their own namespaces separate from the rest of the program—a variable defined in a function only exists within that function.

Nested function calls—When one function calls another function. This is a very common occurrence in practice.

Pure—A function is pure if it does not modify its arguments and has no effects—just a return value.

Shadow—To create a new variable under the same name as an existing variable, making it impossible to refer to the existing variable. If a variable in a function is created under the same name as a variable outside a function, the function-level variable will shadown the outside variable (assuming the global keyword is not used).

Stack overflow—When a function has too many levels of recursion, causing Python to crash.

Chapter 12

Files and Directories

By this juncture, you are already quite familiar with text files. In Section 4.4, we introduced the concept of a text file, granting you the power to store information permanently and your programs the ability to remember data across multiple executions. In Section 8.4, we designed a technique for saving a list of strings to a file, making it possible to consume and generate large datasets. In Section 10.4, we explored the JSON format, a way of turning any Python datastructure into a string and saving it to a file. As far as text files go, your working knowledge is complete.

In this chapter, you will build on that foundation, learning to manipulate the much larger world of files and directories that inhabit your comptuer's hard drive. So far, we've only discussed text files. What about other types of data? How do you read or write to picture files or PDF files? In addition, we've only worked with files stored in the same directory as your Python programs. How do you control files in other locations? What are these "locations" exactly, and how do you create, modify, and delete them?

Files are a common data-storage and data-transmission language that transcends programs, programming languages, and computers. By storing data in a file, you can make it available to later executions of the same program and to other programs that know how to read back the information you've written. These other programs need not be in Python—they could be in JavaScript, C++, Java, or any other fully-featured programming language that knows how to interact with files. And these other programs need not even be on the same computer—files can be transferred to another machine manually or over the Internet.

In this chapter, we will answer all of those questions, giving you the fully ability to interact with your computer's file storage system. By the end of this chapter, you will understand both the way your computer stores and organizes data and how to use Python to interact with this infrastructure. This knowledge, in turn, will give you all the background you need to learn to work with specialized files like PDF documents and spreadsheets; we will explore doing so in the second part of this book.

We will begin this chapter by studying files in full detail, reviewing your existing knowledge of text files and exploring ways of manipulating files that contain non-text data. Afterwards, we will discuss directories, which provide the structure necessary to organize the files on your hard drive. By the end of this chapter, you will be able to write programs that discover, manipulate, create, and delete files and directories, giving your programs control over the full suite of data-storage capabilities that your computer provides.

12.1 Files

Since you have already encountered files several times throughout this book, we will keep our review brief. A file is a piece of information that has been stored permanently on your computer's hard drive. Files exist independent of any program on your computer, and your computer holds onto them indefinitely until you choose to delete them or the physical hard drive breaks.

Like variables, each file has a name to distinguish it from other files and to give you a way to refer to it. File names typically have two parts separated by a dot; examples include `program.py`, `opinion.pdf`, `brief.docx`. The part after the dot is known as the file's extension. An extension tells your computer how to interpret the data stored inside the file. An extension of `txt` typically signifies a text file, `py` a Python file, `docx` a Microsoft Word file, etc. All files store information in binary—a series of 0's and 1's, the basic building blocks of all digital information. A file's extension tells your computer how to translate those 0's and 1's into meaningful information like text, pictures, documents, music, and videos. In a text file, those 0's and 1's are interpreted as letters and numbers. In a picture, they're interpreted as the colors of each individual pixel of the image. Without extensions, your computer would have no way to distinguish between the 0's and 1's that should be interpreted as text and those that should be interpreted as images.

A word of warning. Extensions are not binding contracts about the kind of data stored in a file; they are conventions that most people and programs follow. You can give any type of file any extension you like. For example, you can save an image with the extension txt or a text file with the extension jpg. So long as you treat the image as an image and the text as text, it doesn't matter which extensions you give your files. You can even make up your own extensions if you like. For example, the extension json is just a different extension for a text file that signifies something special about the way the content of the file is structured. However, although you have the freedom to give files any extension you choose, it is good style to follow conventions as closely as possible. That way, the files you create will be easy to share with others.

12.1.1 Text Files

This section should largely be review of Sections 4.4, 8.4, and 10.4. A text file is a particular kind of file where the underlying 0's and 1's represent letters, numbers, and other symbols that you can type on your keyboard. In other words, text files store strings.

Writing. The workflow for writing to a text file (and creating it if it doesn't already exist) consists of three steps.

1. Open the file using the open function. The open function takes two arguments: the string name of the file and a string representing which mode the file is to be opened in. The mode determines whether the file is open for reading, writing, or appending. To open the file in write-mode, the second argument of open should be the string 'w'. Recall that, whenever you open a file in write-mode, the old contents, if any, are wiped out and you start again with a blank file.

   ```
   >>> fh = open('data.txt', 'w')
   ```

   This statement asks Python to open in write-mode ('w') the file called open.txt in the same directory as your Python file.

   Files are under the control of your computer's operating system. The open function asks your operating system to grant Python access to the specified file. The open function returns a filehandle, a special Python type that gives your program the ability to send the operating system commands for manipulating a file.

2. Once you have obtained an open filehandle in write-mode, you can write data using the filehandle's write method. The write method takes one argument—a string to be written to the file. That string is added to the end of the file's current content. Returning to our example, the statement below tells the operating system to write the string 'hello' to the file we opened in step 1.

   ```
   >>> fh.write('hello')
   ```

   The file now consists of the string 'hello'. If you call write more than once while a file is open, Python will concatenate each argument onto the end of the file.

   ```
   >>> fh.write(' world')
   ```

   The file now consists of the string 'hello world'.

   ```
   >>> fh.write('!')
   ```

   After this last call to write, the contents of the file are the string 'hello world!'.

3. When you are done writing, you should close the file by calling the close method on the filehandle with no arguments.

   ```
   >>> fh.close()
   ```

   Doing so tells the operating system that you are done with this interaction with the file. If you neglect to close a file after writing to it, it is possible that some of the information you tried to write won't end up being saved.

Appending. You will often wish to add data to the end of an existing file without wiping it out each time you try to open it. You can open a file in append-mode by passing `'a'` rather than `'w'` as the second argument to open. In append mode, the file is created if it does not exist, but—if it already exists—it is opened unmodified. Every time you write a string, it will be added to the end of the file.

Reading. You can read the contents of a text file from your hard drive into a Python string using a similar three-step process.

1. Open the file. To open it in read-mode, pass open the string `'r'` as its second argument. Opening a file in read-mode will not modify the file in any way.

   ```
   >>> fh_read = open('data.txt', 'r')
   ```

   Since reading from a file is so common, you can leave off the second argument of open completely and Python will assume you intended to open the file in read-mode. In lieu of the previous statement, we could have written

   ```
   >>> fh_read = open('data.txt')
   ```

   and Python would still have opened the file in read-mode.

2. Read the contents of the file into a string by calling the read method with no arguments on the filehandle.

   ```
   >>> data = fh_read.read()
   >>> data
   'hello world!'
   ```

   Depending on the exact structure of the file, you can then process this string further. For example, if it is a JSON-formatted string, you can deseralize it back into a Python datastructure using the `json.loads` function.

3. Close the file. Closing a file is less essential when you are reading from it, but doing so is good style.

   ```
   >>> fh_read.close()
   ```

12.1.2 Binary Files

Many of the files we have worked with so far, like txt files, py files, and json files are text files. You can use the steps in the previous section to create, read, or modify any of these kinds of files. However, most files you will encounter contain something other than a string. An image file (jpg, gif) contains the colors of each individual pixel of the image. A music file (mp3) contains the frequencies to play at each point in time during a song. A PDF file (pdf) contains all of the details of a PDF document, including pictures, colors, text, and formatting. Videos, word processing documents, and many other types of files have their own special-purpose ways of representing this data within files.

There is one common denominator among all kinds of files, text and otherwise: the information they store must somehow be converted into 0's and 1's—binary. Each of the characters in a text file is represented by a number, and these numbers are converted into binary. Each of the pixels in an image is represented by the intensities of the red, green, and blue components of its colors, each of which is stored in binary. The frequencies of a song are numbers that are stored in binary.

Unfortunately, other than text, none of this binary data is innately intelligible to Python. To make sense of the binary underlying an image, you will need to use a special Python library designed for that purpose. The same is true for music, PDF documents, and all other non-text files. We will explore several libraries for manipulating these types of files in the second part of the book. And, although Python cannot make sense of this binary data, it can still read it from a file and write it back to another file. This capability makes it possible to create copies of files, even when you don't know how to interpret the meaning of the data you're reading and writing.

To manipulate binary files, you can use the same tools as for text files: the open function and the read, write, and close methods. When reading or writing a file as binary, pass the open function the flag `'rb'` (for "read in binary-mode") and `'wb'` (for "write in binary-mode"). Suppose we had a picture file, picture .jpg, that we wanted to make a copy of. First, we would need to open the file for reading in binary mode.

```
>>> fh = open('picture.jpg', 'rb')
```

Once the filehandle has been opened, we can read the contents of the file into a variable.

```
>>> picture = fh.read()
```

When we read from a text file, the read method returned a string. When we read from a binary file, it will return a special Python type called a byte string. A byte is eight bits—eight binary 0's and 1's; for convenience, binary information is typically grouped into bytes. A byte string is similar to a string, except that it contains bytes rather than characters. For our purposes, byte strings are a useful way to temporarily store binary data when we are moving it from one binary file to another. Unfortunately, there isn't much we can do with a binary

string, so we won't study ways of manipulating this binary data directly. All we can do with this binary data is write it back to another file.

Returning to the task at hand, we can now close the filehandle since we are done reading the file.

```
>>> fh.close()
```

Now that we have extracted the contents of the original file, we can make a copy. First, we need to open a filehandle in binary write-mode for the file where we wish to store the copy.

```
>>> fh_copy = open('picture_copy.jpg', 'wb')
```

Next, we write the content of the variable picture—the byte string containing the content of the original file— to the copy.

```
>>> fh_copy.write(picture)
```

Finally, we close the filehandle.

```
>>> fh_copy.close()
```

To summarize, we opened the file picture.jpg in binary read-mode, treating its contents as arbitrary, mysterious 1's and 0's. We used the read method to extract the binary from the file and store it in a byte string—the Python datatype used for holding onto this binary data. We then opened picture_copy.jpg, the file where we wanted to store a copy of picture.jpg, in binary write-mode and used the write method to store the content of the byte string in that file. In doing so, we successfully copied made a copy of picture.jpg called picture_copy.jpg.

12.1.3 Summary

At this point in time, you know everything you need to about manipulating files: how to open a file to create a filehandle, read or write, and close it afterwards. For our purposes, text files are particularly convenient. We can manipulate these files directly, reading their contents as strings or or writing strings of our choice. Binary data is more challenging. The translation from binary data into an image or a PDF document is far more complicated. Some of these translations are even proprietary, known only to the program that uses the file. Although binary data is less malleable, the same sequence of operations—open, read/write, close—nonetheless allows you to access and transfer binary data.

12.2 The Filesystem

12.2.1 Introducing Directories

When it comes to the way data is stored on your computer's hard drive, files are merely the smallest building block. Your computer has hundreds of thousands of files, both those that you've created and the ones applications automatically

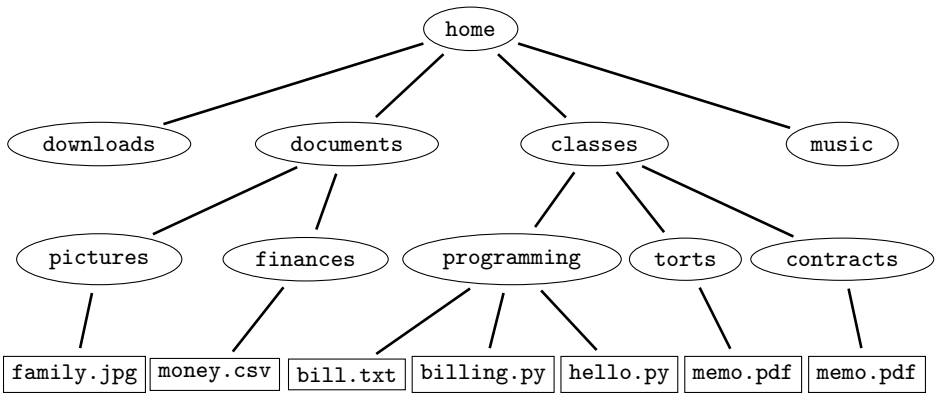
create to keep track of the data they need to operate. Now that you have a firm grasp of files, we will explore the structures used to keep these massive quantities of data organized on your hard drive.

Files are stored in many different "locations" on your computer. For example, you might keep your photos in a location called pictures and your word processing files in a location called documents. Each application you install creates one or more locations on your computer to store the information it generates.

Each of these locations is known as a folder or directory; going forward, we will stick to the term "directory." A directory is a location on your computer that can store files and other directories. For example, your classes directory might store memos you have recently written for school (i.e., files) and directories called contracts (storing files for your contracts class) and torts (storing files for your torts class). In this fashion, directories are recursive—a directory can store other directories, which can, in turn, store other directories. Along the way, each of these directories could also store files. This chain of directories can't go on forever—eventually, directories at the bottom store just files (or nothing at all).

Looking upwards, every directory is inside of some other directory. Your torts directory is inside of your classes directory, which might be inside of your home directory. This chain of direcories can't go up forever either. If you go up high enough in any chain of directories, you'll reach the very top. For reasons that will become clear in a moment, the highest-level directory—the directory from which all other directories originate–is called the root.

To make this all of these abstract notions more concrete, consider the example below.



This diagram contains the files and directories stored on an example hard drive, known collectively as a computer's file system. Files are represented by rectangles and directories by ovals. At the very top of this filesystem is a directory called home. This directory stores four other directories inside: downloads, documents, classes , and music. When considering the relationship

between these four directories and home, home is said to be the parent and downloads, documents, classes, and music are said to be the children.

The downloads and music directories are empty, while documents and classes have children of their own. The documents directory has two children, the finances and pictures directories. The pictures directory has one child—it stores the file family.jpg. The finances directory likewise stores the file money.csv.

We can analyze the classes directory, the sibling of documents, in a similar manner. It has three children, the directories programming, torts, and contracts. The programming directory contains three files: two Python programs (billing.py and hello.py) and a text file (bill.txt). The torts directory contains a single PDF file (memo.pdf). The contracts directory also contains a single PDF file called memo.pdf. Although there are two files called memo.pdf, these are distinct files that just happen to share the same name. You doubtlessly have many files and directories on your computer with the same name but in different locations.

Duplicate names are perfectly acceptable under one condition: a directory cannot contain any immediate children with duplicate names. In other words, torts could not contain a second file called memo.pdf. Likewise, classes could not contain another directory called torts. However, classes could contain a file called memo.pdf, since that name would not conflict with the names of any of its existing children (`programming`, `torts`, and `contracts`). Take a moment to review those three situations one more time to ensure you fully understand the rule.

Collectively, the filesystem as laid out in the diagram is often referred to as a tree. This is why the top of the filesystem—the home directory's parent—is known as the root: it is the source from which all of the branches (each of the intermediate directories) flow. These branches contain leaves—the files at the ends of the branches.

Although this diagram is a simplified version of a real filesystem, every single file and directory on your hard drive is organized into one large tree structured as portrayed in the diagram above. In the sections that follow, we will learn to use Python to traverse, manipulate, and rearrange this tree.

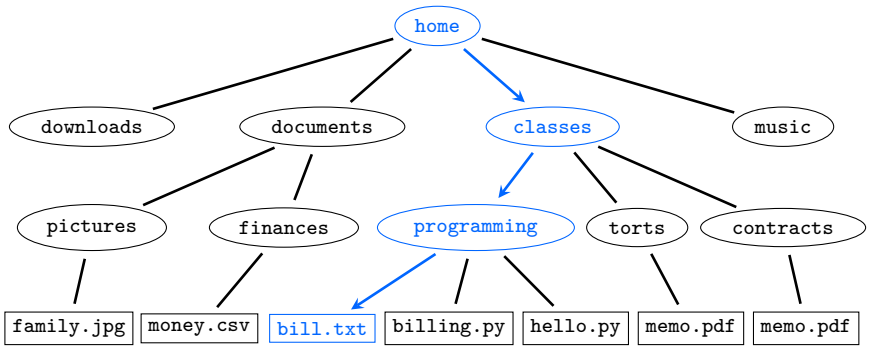
12.2.2 Naming

In order to access or manipulate a file or directory, you have to have a way to refer to it. In this section, we will explore the mechanics of naming a file or directory, which is the first step toward being able to operate on it.

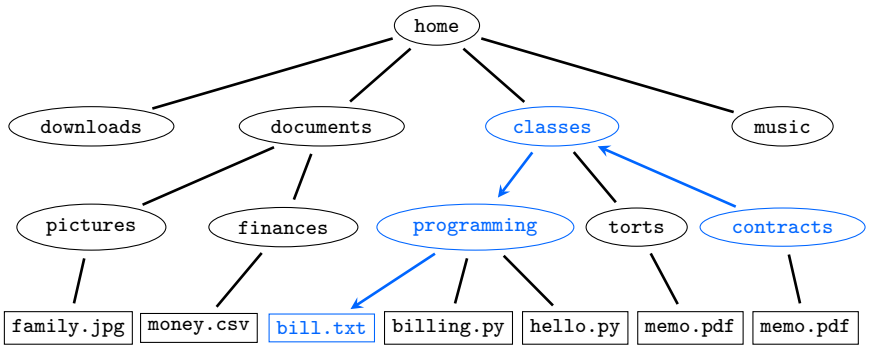
When it comes to naming files, everything is relative. Suppose we are trying to determine how to refer to the file bill.txt. As we discovered earlier when we encountered two files with the same name, the name alone is not enough to uniquely distinguish a particular file. We could start by trying to describe its location relative to other parts of the tree. For example, "the `bill.txt` that is within the directory called programming." But, as we learned earlier, there could be two directories called programming in different locations on the filesystem, each with a child file called bill.txt.

We can repeat this process, going upwards, level by level, until we've reached the very top of the tree. At this point, we would describe the file we want as "the file `bill.txt` within the directory programming within the directory classes within the directory home at the root of the tree." This name is guarnateed to be unique. Since no duplicate names are allowed in the same directory, there can't be another directory called home at the root of the tree. Within this directory, there can't be a duplicate directory called classes, which can't contain a duplicate directory called programming, which can't contain a duplicate file called bill.txt.

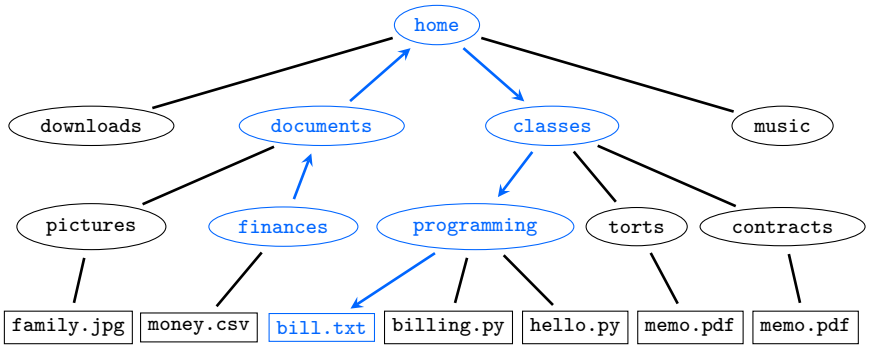
In summary, you can uniquely identify a file or directory by referring to its chain of parents and ancestors all the way to the root of the filesystem. This chain is known as a path—a set of directions to get from one part of the filesystem to another. The kind of path we've described here is specifically concerned with getting from the root of the filesystem to the file or directory we want to name. This path takes the form of the directions, "start at the root, then the home child directory, then the classes child directory, then the programming child directory, then the file bill.txt." A path that starts at the root is called an absolute path, since it describes the location of a file in relation to the root—the center of the filesystem's universe. Highlighted in blue below is the absolute path from the root to bill.txt.



Absolute paths aren't the only way to name a file uniquely. In fact, we can start from any location we want in the filesystem, so long as we're clear about where we start. These paths are known as relative paths, since they describe file locations relative to some other location on the filesystem. For example, suppose we started inside contracts directory and wanted to describe the path to bill.txt. First, we would need to go up one level, so that we are in the classes directory. Then we need to go down a level to the programming directory. Finally, we need to select the file called bill.txt. On the diagram:



What if we started inside the finances directory? We would need to go up a level to inside the documents directory. Then we would need to go up a level again to inside the home directory. Then we could go down to the classes directory, down again to programming, and finally down again to bill.txt. Visually:



What happens if our relative path starts from the same directory that it ends at? For example, what if we are already starting inside the programming directory? The path is very short, just "pick the file called bill.txt."

12.2.3 Building Paths in Python

Absolute paths. Now that we understand how the filesystem is structured and the concepts of naming files via absolute and relative paths, we are ready to put this knowledge to work in Python. In Python, a path is simply a string. For example, the absolute path from the root to the file bill.txt could be written as the string

```
'/home/classes/programming/bill.txt'
```

The / at the beginning indicates that we are starting the path from the root. The rest of the path contains the sequence of directories we will need to follow to get from the root of the filesystem to bill.txt; each of these directories is separated with a slash. First, we express that we wish to go inside the home directory by writing the name home directly after the first slash.

'/home'

We then go into the classes directory within the home directory, meaning that the path gets updated to

'/home/classes'

Going inside the programming directory, the path gets updated to

'/home/classes/programming'

Finally, we select the file bill.txt within that directory.

'/home/classes/programming/bill.txt'

Before continuing any further, we will note particularly frustrating aspect of constructing paths in Python. This path will only work on MacOS and Linux computers. In MacOS and Linux, the path separator (the character between each step in the path—the / above) is a forward slash /; on Windows computers, it is a backslash \. In MacOS and Linux, the root directory is written as a single forward slash at the beginning of the path; on Windows computers, the root is written as the drive letter where the files are stored, for example, `C:\`. In other words, the same path would be written on Windows as

'C:\home\classes\programming\bill.txt'

The lesson here is that, if you try to hand-code the names of paths, you can't do so in a way that will simultaneously work on MacOS, Linux, and Windows. However, it is always good style to write programs that are platform-independent—that work properly no matter what operating system the user is running. To make it possible to write platform-independent paths, Python provides a set of helpful path-manipulation tools. To make use of them, you will need to import a Python library called os that contains a set of functions for manipulating paths, files, and directories.

```
>>> import os
```

The first function we will learn is called os.path.join. This function takes as many string arguments as you give it. It will join those string arguments together into a path using the appropriate kind of slash for whatever operating system the program is running on (forward slash for MacOS and Linux, backslash for Windows). For example, suppose we wanted to refer to the directory finances inside the directory documents. We would write

```
os.path.join('documents', 'finances')
```

which would evaluate to the string 'documents/finances' on MacOS and Linux and 'documents\finances' on Windows. You can directly access the correct kind of slash for a particular operating system using the variable os.sep, which is '/' on MacOS and Linux and '\' on Windows.

Returning to our previous example, how would we use os.path.join to create the absolute path to bill.txt? The path starts with home, then classes, then programming, then arrives at bill.txt.

```
os.path.join('home', 'classes', 'programming', 'bill.txt')
```

This statement will evaluate to the string

```
'home/classes/programming/bill.txt'
```

on MacOS and Linux and

```
'home\classes\programming\bill.txt'
```

on Windows. As the above example illustrates, the os.path.join function makes it easy to write platform-independent paths. Unfortunately, the path we created isn't quite right. We want an absolute path, meaning the path should start from the root. In other words, we want a leading / at the beginning of the MacOS and Linux path and a leading C:\ at the beginning of the Windows path. Python will begin the path in the proper way to create an absolute path if we pass os.sep (the correct kind of slash for our OS) as the first argument to os.path.join.

```
os.path.join(os.sep, 'home', 'classes', 'programming', 'bill.txt')
```

This statement will do what we want, generating the path

```
'/home/classes/programming/bill.txt'
```

on MacOS and Linux and

```
'C:\home\classes\programming\bill.txt'
```

on Windows. You should only use this trick when you want to create absolute paths. When creating relative paths, you should leave off os.sep at the beginning of the path since relative paths start do not start at the root. We will discuss relative paths in more detail in a moment.

The os.path.join function can also be used to join two paths that already exist; its inputs need not be individual files or directories. For example, on MacOS or Linux, we could write

```
os.path.join('/home/classes', 'programming/bill.txt')
```

to produce the path

```
'/home/classes/programming/bill.txt'
```

At its simplest level, os.path.join will simply insert the value of os.sep between each of its arguments.

Relative paths. Absolute paths always proceed from the root of the filesystem downwards, level by level, to the desired file or directory. As we learned previously, a slash (forward or backwards, depending on the operating system) expresses that we want to take a step downards. For example, the path '/home/classes' expresses that we wish to start at the root of the filesystem, then take a step downwards into the home directory, then take a step downwards again into the classes directory.

Relative paths can be more complicated: they can go both up and down. Suppose that we wished to express the relative path to bill.txt starting from

inside the directory finances. We will need to go up twice, first from inside finances to inside documents and then from inside documents to inside `home`. Only then do we proceed dowards into classes and programming before arriving at our destination. To express going up one level in a path, write the string `'..'` . This string works universally across both MacOS, Linux, and Windows.

Let's step through constructing the path that starts inside the finances directory and ends with the file bill.txt. First, we step up one level, from inside of finances to inside of documents. The path for doing so is:

```
os.path.join('..')
```

This path tells your computer to "go up one level." Next, we step up another level, from inside of documents to inside of home. To express that, we add another appearance of `'..'` .

```
os.path.join('..', '..')
```

So far, this path tells your computer to "go up one level then go up another level." Now that we are inside of the home directory, we can proceed downwards. First, we enter the directory classes.

```
os.path.join('..', '..', 'classes')
```

Then, we enter the directory programming.

```
os.path.join('..', '..', 'classes', 'programming')
```

Finally, we can name the file itself.

```
os.path.join('..', '..', 'classes', 'programming', 'bill.txt')
```

In summary, we had to go up into documents and up again into home, after which we went down twice into classes and programming. On MacOS and Linux, this expression will evaluate to the string

```
'../../classes/programming/bill.txt'
```

On Windows, it will evaluate to

```
'..\..\classes\programming\bill.txt'
```

Notice that the MacOS and Linux path does not begin with a forward slash nor does the Windows path begin with the string C:\. Relative paths are written relative to a particular location on the filesystem (in this case, the finances directory), so they do not begin at the root.

The current working directory. You now have all of the tools you need to name any file or directory on your computer using Python. However, one practical question remains: when you write relative paths in a Python program, where do those paths start from? To answer that question, you will need to know about the concept of a current working directory.

Every time you execute a Python program, it computes all relative paths from the location where your shell had navigated to when you caused the

Python program to execute. For example, if you navigated your shell to the programming directory and typed in the command

```
$ python3 billing.py
```

All relative paths in billing.py would be computed starting from the `programming` directory. If you wanted to access the file money.csv, you would need to use the relative path

```
os.path.join('..', '..', 'documents', 'finances', 'money.csv')
```

This path goes up one level (from inside programming to inside classes), up another level (from inside classes to inside home) and then down into documents, finances, and finally to money.csv.

What about the relative path to bill.txt? Since it is in the same directory as that from which we executed billing.py, the relative path is simply 'bill.txt'—we're already in the right directory. This should explain why, in previous chapters, we only referred to files in the same directory as our Python programs. We hadn't yet introduced the concept of paths, so the only relative paths we could write were those naming files in the same directory as our programs.

The directory that serves as the source of a program's relative paths is known as its current working directory. As this name suggests, we will see soon that the current working directory can be changed. To access the current working directory for a particular program, you can use the os.getcwd function. This function, which takes no arguments, returns a string containing the absolute path to the program's current working directory. If the program billing.py were executed from the programming directory, the expression

```
os.getcwd()
```

would evaluate to the string '/home/classes/programming' on MacOS and Linux and 'C:\home\classes\programming' on Windows.

You can change the program's current working directory by using the os.chdir function, which takes a single string argument containing a path. This function will change the program's current working directory to the location referred to by the path. This path could be an absolute path (in which case the current working directory will change to the location it references) or a relative path (in which case the current working directory will change to the location specified by starting from the old current working directory and following the relative path). For example, executing the statement

```
os.chdir('..')
```

in billing.py would cause Python to move the current working directory up one level, from programs to classes. Subsequently executing the statement

```
os.chdir(os.path.join('..', 'documents'))
```

would move the current working directory up one level (to home) and then down into documents. Your program could then refer to the file family.jpg using the relative path

```
os.path.join('pictures', 'family.jpg')
```

moving down one level into pictures and then to family.jpg. To refer to bill.txt, it would need to use the relative path

```
os.path.join('..', 'classes', 'programming', 'bill.txt')
```

moving up one level (from inside documents to inside home) and then down into classes and programming before arriving at bill.txt.

Technically, you never need to use os.chdir to change the current working directory. Using relative paths, you can access any location on the filesystem from any other location on the filesystem. However, os.chdir can come in handy for two reasons. First, if you are accessing many files in a far away directory, it may be easier to use os.chdir once to switch into that directory and then refer to each file using a short relative path than to use a long relative path many times. Second, someone could start your program from somewhere unusual, causing you to end up with an unexpected current working directory; the os.chdir command can be used to ensure that your program always resets itself to the right current working directory so that relative paths are computed properly.

The payoff of this section is that you can now use the open function to open a file anywhere in the filesystem by providing it the appropriate relative path (as compared to the program's current current working directory) or the appropriate absolute path. For example, if a program's current working directory is programming, writing the statement

```
fh = open(os.path.join('..', '..', 'documents', 'finances', 'money.csv'))
```

will open the file money.csv (referred to using its relative path) in read-mode and store the corresponding filehandle in the variable fh.

12.3 Exploring the Filesystem

In the previous section, we learned how your computer's files are organized, how to name specific files, and how to open (and thereby create, read and write) files anywhere on your hard drive. In this section, we go one step further, giving you the ability to explore the filesystem and discover files.

In our running example from the previous section, we worked with a filesystem whose layout we knew ahead of time. We knew that the root directory had one child directory (home) that had four children of its own (downloads, documents, classes, and music), and we knew the exact path from the root to the file bill.txt. In reality, your Python programs will begin with no prior knowledge of the structure of the filesystem. They could be running on a different computer with a different set of files and directories or even on a different operating system with entirely different conventions about how files are organized. In order to ensure your programs are able to adapt to whatever circumstances they find themselves in, they will need to be able to explore and map out the filesystem.

The os library provides several useful functions that will make this job easier. In this section, we work through a series of example scenarios in which you need to use these functions. These scenarios are not contrived—each one involves a step you are likely to need to perform often in the real world.

12.3.1 Checking that a File Exists

When a program tries to open a file that doesn't exist, it will crash. For example, consider the program below, which asks the user for a name of a file and prints out the number of words that it contains.

```
1  # Ask the user for the name of the file.
2  file = input('Name of file: ')
3
4  # Open and read the file.
5  fh = open(file)
6  contents = fh.read()
7  fh.close()
8
9  # Count the number of words in the file.
10 words = len(contents.split())
11 print('The file {0} contains {1} words.'.format(file, words))
```

First, the program asks the user to provide the name of a file (line 2). The program then reads the contents of the file into a string (lines 5-7). Line 10 uses the split method to divide contents into a list of its constituent words and the len function to determine the size of that list—the number of words in the file. Finally, line 11 prints this information back to the user. For the purposes of this example, we have saved this file in the programming directory under the name word_count.py.

Suppose we want to use this program to count the number of words in bill.txt. What string should we pass as input to the program? It depends on the program's current working directory. We should pass in a string containing a relative path from the program's current working directory to bill.txt. Recall that a program's current working directory is the directory to which your shell has navigated when it executes the program.

First, we open up our computer's shell. This might be the terminal in MacOS or Linux and PowerShell in Windows. The examples below will appear as if performed on MacOS or Linux, but the commands will work identically in PowerShell. First, we use the pwd command, which asks the computer to display the directory that the shell is currently navigated to. (Note that the $ character is meant to represent all of the text your shell displays before allowing you to type commands. Your computer will probably have more text there.)

```
$ pwd
/home
```

Our shell is currently navigated to the home directory inside of the root directory. To execute word_count.py, we will need to navigate the shell into the directory where it is stored (programming). We do so with the cd command, which navigates the shell from its current directory to a particular path. You can provide cd the same kinds of paths as you would present to Python.

```
$ cd classes/programming
$ pwd
/home/classes/programming
```

Starting from the home directory, we navigated into the classes directory and, within that, the programming directory. Finally, we called pwd to confirm we ended up in the right place. We can finally execute the program.

```
$ python3 word_count.py
Name of file: bill.txt
The file bill.txt contains 29 words.
```

Since the shell was navigated to the /home/classes/programming directory, that became the program's current working directory. We passed it the path bill.txt since that file is inside the current working directory. The program behaved as expected, printing the number of words in the file.

What happens if we accidentally pass the program a file that doesn't exist?

```
$ python3 word_count.py
Name of file: brief.txt
Traceback (most recent call last):
  File "word_count.py", line 5, in <module>
    open('brief.txt')
FileNotFoundError: No such file or directory: 'brief.txt'
```

Unsurprisingly, the program crashes. The error message is quite clear: the program produced a FileNotFoundError because the file brief.txt doesn't exist.

As we have learned, it is bad style for a program to crash when it is given bad input. Instead, it should fail gracefully, printing a helpful error message for the user. The Python os library contains several functions that will be useful to us here. First, it includes the function os.path.exists, which takes as its sole argument a string containing a path. It returns the boolean value True if the file or directory to which the path refers actually exists and False otherwise.

We can use the os.path.exists function to avoid the FileNotFoundError by checking whether the file exists before trying to open it. If it does exist, we can open the file and complete the program as before. If not, we can print a helpful error message and end the program.

```
1 import os
2
3 # Ask the user for the name of the file.
4 file = input('Name of file: ')
5
```

```
6  # Check whether the file exist.s
7  if os.path.exists(file):
8      # Open and read the file.
9      fh = open(file)
10     contents = fh.read()
11     fh.close()
12
13     # Count the number of words in the file.
14     words = len(contents.split())
15     print('The file {0} contains {1} words.'.format(file, words))
16 else:
17     print('Sorry, file {0} not found.'.format(file))
```

This program is largely the same as before with two key differences. First, we imported the os library on line 1 so that we can access the functions it provides; imports should always appear at the very top of the program. On line 7, the program uses the os.path.exists function to check whether the path stored in the variable file refers to a valid item in the filesystem. The if-statement on line 7 uses the result of that function call as its boolean test. If the path does exist, it executes the rest of the program as before. If it doesn't exist, the else-statement on lines 16 and 17 executes instead, printing out a helpful error message. If we execute the program with the same nonexistent input as before, the program now avoids crashing.

```
$ python3 word_count.py
Name of file: brief.txt
Sorry, file brief.txt not found.
```

Unfortunately, we're not quite done yet. There is one more way that a clumsy user could cause this program to crash. What if, rather than providing a file as input, a user provided the name of a directory? The call to the os.path.exists function on line 7 will return True, since the path name really does exist, just as a directory rather than a file. On line 9, the program will then try to open the directory as if it were a file. Since the open function can only open files, Python will crash.

```
$ python3 word_count.py
Name of file: ../torts
Traceback (most recent call last):
  File "word_count.py", line 7, in <module>
    fh = open(file)
PermissionError: [Errno 13] Permission denied: '../torts'
```

In the absence of the preceding discussion, this error message would be quite difficult to decipher. Python produces a PermissionError stating that permission to open ../torts was denied. As we now know, this is Python's enigmatic way of conveying that you passed the open function a directory rather than a file.

Instead of crashing, your program should print a clear error message. The os library includes two other helpful functions that test whether a particular path is a file (os.path. isfile) or a directory (os.path. isdir). If the provided path is a file, os.path. isfile returns True; if it is anything other than a file (such as a directory or a path that doesn't exist) it returns False. The os.path. isdir behaves analogously for directories. Using these functions, we can modify our program to print a separate error message when it encounters a directory.

```
1  import os
2
3  # Ask the user for the name of the file.
4  file = input('Name of file: ')
5
6  # Check whether the path is a file.
7  if os.path. isfile (file):
8      # Open and read the file.
9      fh = open(file)
10     contents = fh.read()
11     fh.close()
12
13     # Count the number of words in the file.
14     words = len(contents.split())
15     print('The file {0} contains {1} words.'.format(file, words))
16
17 # Checks whether the path is a directory.
18 elif os.path.isdir(file):
19     print('{0} is a directory, not a file.'.format(file))
20
21 # Otherwise, the path doesn't exist.
22 else:
23     print('Sorry, file {0} not found.'.format(file))
```

On line 7, this program calls the os.path. isfile function (rather than os.path .exists as before) to check whether the provided path is a real file. If so, the program executes lines 8-15 and prints the number of words in the file. Otherwise, the program proceeds to the elif statement on line 18, which uses the os.path.isdir function to check whether the path is really a directory. If so, it prints an error message to that effect on line 19. Finally, if the path refers to neither a file nor a directory, then it must not exist. The else statement on line 21 prints an error message accordingly (line 22).

Retrying the input that caused the program to crash before, the far clearer error message on line 19 prints instead.

```
$ python3 word_count.py
Name of file: ../torts
../torts is a directory, not a file.
```

The lesson of this exercise is that, whenever your program attempts to open a file that may not exist, it should first check to make sure the file does, in fact, exist. In many situations, your program will indeed know whether a file exists. However, in this example, the filename was provided as user input, so your program had no prior knowledge about whether the file existed. In other cases, your program will retrieve the name of an unfamiliar file from elsewhere, perhaps even reading it in from some other file. Whenever you are uncertain, you should perform this check to prevent your program from unexpectedly crashing.

Some of your programs will require a specific file to exist in order to run properly. For example, a billing program might expect to work on a file called bill.txt containing a string representing the total amount that has been billed. However, the first time the program executes, the file bill.txt may not exist, so the program should create it and set the initial balance to $0. If the file already exists, however, the program should absolutely not create a new version of the file containing a balance of $0; it should open the file and read back the current balance. We can express this logic in Python using os.path.isfile.

```
1  import os
2
3  # Check whether bill.txt already exists.
4  if os.path.isfile('bill.txt'):
5      # If it does, open the file and read back the balance.
6      fh = open('bill.txt')
7      balance = int(fh.read())
8      fh.close()
9  else:
10     # If it doesn't, create it and initialize the balance $0.
11     fh = open('bill.txt', 'w')
12     fh.write('0')
13     fh.close()
14     balance = 0
15
16 # The rest of the program goes here.
```

As always, the program begins by importing the os library. On line 4, it tests whether the file bill.txt exists and is a file. If so, it opens the file in read-mode (line 6), retrieves the string balance and converts it to an integer (line 7), and closes the file. Alternatively, if the file doesn't exist, it opens the file in write-mode (line 11), writes the initial balance of $0 (line 12), closes the file (line 13), and sets the variable balance to 0.

Whether the file bill.txt already existed or needed to be created for the first time, the program is now ready to compute with the value. This strategy for initializing a file if it doesn't already exist is very common in practice.

12.3.2 Exploring the Current Directory.

In the preceding section, we tested whether a specific file existed. Here, we will gain the ability to discover new files and directories by exploring the filesystem. Suppose that your program has been placed into a directory with numerous text files representing individual bills. Each of these text files contains a string with the amount of money billed. To distinguish these files from other miscellaneous text files, they have been given the extension `.bill`. Recall that we can give a file any extension we like so long as we know how to treat its contents. In this case, we know that files ending in `.bill` are really just text files.

The program's goal is to compute the total of all of these smaller bills. To do so, the program will need a way to figure out the names of all of the files in the current working directory. Python provides a function called `os.listdir` to do just that. (Note that the name of this function is just `os.listdir`—it does not contain the name path.) The `os.listdir` function takes no arguments and returns a list containing the names of all of the files and directories within the current working directory. If we called this function in the home directory, it would return the list `['downloads', 'documents', 'classes', 'music']`—the four child directories of home. If we called it within the programming directory, it would return the list `['bill.txt', 'billing.py', 'hello.py']`. The `os.listdir` function only returns the immediate children of the current directory; it does not return any grandchildren or other descendents.

Returning to the task at hand, our program can use `os.listdir` to get a list of all of the files in the current working directory. It will then have to select only those files that end in the extension `.bill`, read the balances, and add those balances together.

First, we import the os library and create a variable to store the total balance.

```
1  import os
2
3  # Keep track of the total balance.
4  total_balance = 0
```

Then, we use the `os.listdir` function to get a list of all of items in the current working directory.

```
6  # Get a list of all items in the current working directory.
7  items = os.listdir()
```

We can now iterate over this list of items using a for-loop.

```
9  # Iterate over the items in the current working directory.
10 for item in items:
```

What exactly should take place inside this for-loop? First, we should examine the item to ensure that (1) it is a file and not a directory and (2) that it ends with the extension `.bill`. We can ignore all other entries.

```
10 for item in items:
11     # Ensure item is a bill file.
```

```
12      if os.path.isfile(item) and item.endswith('.bill'):
```

This if-statement contains two conditions. First, it uses the os.path.isfile function to ensure that item is indeed a file. Second, it uses the string endswith method to ensure that the last characters of the filename are the extension '.bill'. Once inside this if-statement, we can rest assured that item refers to a billing file. We can now read the balance from the file and add that value to the total balance.

```
10  for item in items:
11      # Ensure item is a bill file.
12      if os.path.isfile(item) and item.endswith('.bill'):
13          # Read the balance from the file.
14          fh = open(item)
15          balance = int(fh.read())
16          fh.close()
17
18          # Add the balance to the total.
19          total_balance += balance
```

On line 14, the program opens the file whose name is stored in item. It reads the balance from the file and converts it into an integer on line 15. After closing the file on line 16, the program adds this balance to the total accumulated balance on line 19. Finally, it prints the total balance:

```
21  # Print the overall balance.
22  print('The total balance is ${0}'.format(total_balance))
```

The os.listdir function is a powerful tool. Where os.path.exists, os.path.isfile, and os.path.isdir made it possible to test the qualities of files already known to your program, os.listdir gives your program the ability to actively explore the filesystem, discovering files and directories on its own.

12.3.3 Exploring All Descendants

Although os.listdir only lists files and directories stored inside the current working directory, you can use this function in combination with os.chdir to explore the entire file system. Every time os.listdir finds a directory, you can use os.chdir to step into that directory, run os.listdir again, and discover all of the files stored inside this child directory. If you begin this process at the root of the filesystem, you can slowly walk your way down the entire tree, discovering every file and directory on your computer.

In fact, you can even do this without ever using os.chdir. The os.listdir function optionally takes a single string argument representing a path. If this path is a valid directory, os.listdir will return the list of that directory's children. For example, if the current working directory were the programming directory, the function call

```
os.listdir(os.path.join('..', '..', 'documents')
```

would return the list ['pictures', 'finances'] —the children of the /home/documents directory.

In this section, we are going to use this capability to write a program that will find every file and directory that is a descendant of a particular directory in the filesystem. Before writing any code, we will discuss the general strategy for writing this program. Initially, our program will begin at a particular starting point in the filesystem, for example the home directory. It will then use the os.listdir function to get this directory's children. It will do the same for each of these children, running os.listdir on downloads, documents, classes, and music in the same fashion. These directories will uncover further directories, for which the program will repeat the same process. Along the way, we will find the files that these directories contain.

Notice that this strategy is recursive—for every directory it comes across, it applies the same strategy, repeating it directory by directory until it has covered every descendant of the starting point. To turn this strategy into code, we will use a list. This list will keep track of all of the directories we have yet to explore—a to-do list of directories. Every time we encounter a new directory, we will add it to the list. Every time we finish processing a directory, we will remove it from the to-do list and explore whichever directory is next on the to-do list. When the to-do list is empty, we will have explored everything.

This program will begin as all of our programs in this section have. First, we will import the os library.

```
1  import os
```

Next, we will ask the user for the name of the directory to explore.

```
3  # Receive the name of a directory as input.
4  start_directory = input('Directory to explore: ')
```

The program should only proceed if the name provided is actually a valid directory.

```
6  # If the input is not a valid directory ...
7  if not os.path.isdir(start_directory):
8      # ... print an error message.
9      print('{0} is not a valid directory.'.format(start_directory))
```

If start_directory does contain the name of a valid directory, we can begin exploring it.

```
11 # Otherwise, explore the directory.
12 else:
```

As we discussed before, our first step is to create a to-do list containing all of the directories we still need to explore. Initially, this list contains only the value of start_directory. It could be that there are no further directories to explore; for example, start_directory could be pictures. However, start_directory could contain many descendant directories that will eventually be added to this to-do list.

```
11 # Otherwise, explore the directory.
12 else:
13     # Create the to-do list of directories to explore.
14     # The first entry is start_directory.
15     todo_list = [start_directory]
```

Now, we need to create a loop. This loop should continue iterating until todo_list is empty, at which point we know we have explored every descendant directory of our starting point. Although for-loops have been our default choice of loop since they were first introduced, this situation calls for a while-loop. A for-loop is useful for iterating over datastructures like lists and dictionaries, but a while-loop is designed to loop indefinitely until a particular condition is met. In our case, we need to loop until todo_list is empty, a perfect fit for a while-loop.

```
16     # Explore each directory.
17     while len(todo_list) > 0:
```

The while-loop's boolean condition says that it should continue iterating so long as todo_list contains more than zero elements, meaning that there are still more directories to explore.

Inside this while loop, we should perform three steps. First, we should remove the next directory from the to-do list. Then, we should use os.listdir to get the list of items in this directory. Finally, we should sift through these items, printing what we find and adding any directories to the to-do list for further exploration.

```
16     # Explore each directory.
17     while len(todo_list) > 0:
18         # Remove the last item from the to-do list.
19         next_directory = todo_list.pop()
20
21         # Retrieve the items in the directory.
22         children = os.listdir(next_directory)
```

Line 19 takes care of the first task, extracting the last item in the list into the variable next_directory. It does so using the list pop method, which removes the last item from the list and returns it. Line 22 takes care of the second task, using os.listdir to get the children of the directory. We are now ready to iterate over these children, which we can do using a for-loop.

```
24         # Iterate over all children of the directory.
25         for child in children:
```

As we examine a particular child file or directory, our first task is to put together its full path. Recall that the list returned by the os.listdir function contains the name of the directory's children, not their full relative paths. For example, if the current working directory was programming and we called

```
os.listdir(os.path.join('..', '..', 'home')
```

it would return the list

['downloads', 'documents', 'classes', 'music']

These are just the names of the children of the home directory, not relative paths from our current working directory (programming). To properly refer to these items, we can use os.path.join to combine these names with the paths that lead to them.

```
24          # Iterate over all children of the directory.
25          for child in children:
26              # Build a path to this child.
27              child_path = os.path.join(next_directory, child)
```

Line 27 combines the path to the parent directory (next_directory) with the name of the child, creating a path to the child and storing it in the variable child_path.

Next, we test whether this path refers to a file or a directory. If it refers to a directory, we should add it to the end of the to-do list.

```
24          # Iterate over all children of the directory.
25          for child in children:
26              # Build a path to this child.
27              child_path = os.path.join(next_directory, child)
28
29              # If it is a directory ...
30              if os.path.isdir(child_path):
31                  print('DIRECTORY: ' + child_path)
32
33                  # Add it to the to-do list.
34                  todo_list.append(child_path)
35
36              # Otherwise, it is a file.
37              else:
38                  print('FILE: ' + child_path)
```

Lines 30 to 34 handle the case where it is a directory. First, line 31 prints a string to that effect. Line 34 then adds this path onto the end of the to-do list so that we can explore it later. Alternatively, if it is a file, line 38 prints a string to that effect. Either way, the nested for-loop then moves on to the next child.

This is the entire program. The outer while-loop keeps exploring new directories until there are none left to explore. The inner for-loop processes the contents of each directory. The exact mechanism by which this program explores the filesystem is a little, so we will consider a few concrete examples.

First, consider what would happen if we ran this program on the `programming` directory. The to-do list will initially contain the programming directory. Since the to-do list isn't empty, the while-loop will execute. Line 19 will pop the programming directory off of the to-do list, leaving it empty. Line 22 will retrieve all of this directory's children: `bill.txt`, `billing.py`, and `hello.py`.

The inner for-loop will iterate over these entries. Since none of them are directories, it will print the names of these files.

```
FILE: bill.txt
FILE: billing.py
FILE: hello.py
```

Once the for-loop has concluded, Python will return to the beginning of the while-loop (line 17). Since the to-do list is now empty, the while-loop will exit and the program will terminate.

What if we instead ran this program on the directory `../classes`? (Assuming that our current working directory is programming, we need to use the `..` to provide a valid relative path to the classes directory.) Initially, the to-do list will contain the string `'../classes'`. Python then enters the while-loop, where it pops that path into the variable next_directory, leaving the to-do list empty. The nested for-loop considers each of the children of the class directory: programming, torts, and contracts. Since each of these children is a directory, paths to these directories are added to the to-do list and the directory names are printed one by one.

```
DIRECTORY: ../classes/programming
DIRECTORY: ../classes/torts
DIRECTORY: ../classes/contracts
```

Once the for-loop has completed, Python returns to the beginning of the while-loop. Since there are now three entries in the to-do list, the while-loop executes again, popping off the last entry, the path `'../classes/contracts'`. The for-loop iterates over the children of the contracts directory, printing the path to memo.pdf.

```
FILE: ../classes/contracts/memo.pdf
```

Since none of the children of contracts are directories, nothing is added to the to-do list. However, there are still two items left in the to-do list (programming and torts), so the while-loop executes again, popping the path `'../classes/torts'`. The for-loop iterates over its children, printing the path to the other memo.pdf.

```
FILE: ../classes/torts/memo.pdf
```

Since torts has no children that are directories, nothing is added to the to-do list. The outer while-loop iterates again, popping the last entry, `'../classes/programming'`, off of the to-do list. The for-loop prints the paths to of each of the three files stored inside this directory.

```
FILE: ../classes/programming/bill.txt
FILE: ../classes/programming/ billing.py
FILE: ../classes/programming/hello.py
```

Since none of these items are directories, nothing is added to the to-do list. When the while-loop performs its boolean check, it finds that the to-do list is now empty and the program finishes.

We could apply similar analysis to any other directory in the filesystem, although doing so for a directory higher in the tree (like home) would be quite lengthy. This example demonstrates the full capabilities of the filesystem-exploration tools we have developed over the preceding pages. Using these techniques, a Python program can explore the entire filesystem of the computer on which it is running, finding and interacting with any files of its choosing.

12.4 File and Directory Management

So far in this chapter, you have learned how to create, read, and modify individual files and to explore the files and directories present on your computer. However, in your day-to-day computer use, you are accustomed to manipulating files in directories in many other ways. You can create directories, move or copy files and directories from one location to another, and delete files and directories. In this section, you will learn how to perform these operations in Python, giving you the full ability to manipulate your computer's filesystem. Many of the functions we discuss in this section are provided by the Python os library. However, another Python library, shutil (short for "shell utilities") will also be helpful. As always, make sure to import a library before you try to use it.

A word of warning. All of the file-manipulation operations you perform in this section are permanent. *If you delete a file or directory, it is deleted permanently. It does not go to the trash, recycle bin, or a temporary location. It is deleted forever.* Many of the operations you will learn can delete files—not just the delete commands. If you move or copy a file to a location where another file already exists, Python will overwrite the existing file, deleting it forever. Be exceedingly careful when using these operations, and test your code on dummy or expendable files before using it on files that are important. *In short, you can cause permanent damage to your computer with these operations, so be very careful.*

12.4.1 Manipulating Files

We will begin by filling in several file-manipulation operations that go beyond creation, reading, and writing. Suppose that we have opened interactive Python and navigated our current working directory to the programming directory.

```
>>> import os
>>> os.getcwd()
'/home/classes/programming'
```

In this directory are three files.

```
>>> os.listdir()
['bill.txt', 'billing.py', 'hello.py']
```

Renaming files. You can rename a file using the os.replace function. It takes two arguments: a path to the existing location of the file and a path to the new location. For example, to rename hello.py to hello_world.py, you would use the statement:

```
>>> os.replace('hello.py', 'hello_world.py')
>>> os.listdir()
[' bill .txt ', ' billing .py ', 'hello_world.py']
```

The files you rename do not have to be in the current working directory. You can rename the file memo.pdf in the torts directory by providing the appropriate paths. First, we create a path to the torts directory from the programming directory (our current working directory). As expected, it contains one file: memo.pdf.

```
>>> torts = os.path.join('..', 'torts')
>>> os.listdir(torts)
[' memo.pdf']
```

Next, we create a path to the old file (the variable old_path) and a path containing the name that we want to assign to the new file (the variable new_path). We use the os.replace method to rename the file accordingly.

```
>>> old_path = os.path.join(torts, 'memo.pdf')
>>> new_path = os.path.join(torts, 'torts_memo.pdf')
>>> os.replace(old_path, new_path)
```

Afterwards, os.listdir reveals that the renaming operation was successful.

```
>>> os.listdir(torts)
[' torts_memo.pdf']
```

Moving files. As far as your filesystem is concerned, renaming a file and moving it to a new location are identical operations. In other words, moving a file is just renaming it so that its new name is in a different directory. For example, suppose we wanted move the file money.csv in the finances directory into the programming directory so that it is convenient for our Python programs to access. First, we need to create a relative path to the old file; again, our current working directory is the programming directory.

```
>>> finances = os.path.join('..', '..', 'documents', 'finances')
>>> os.listdir(finances)
[' money.csv']
>>> old_path = os.path.join(finances, 'money.csv')
```

The first command creates the path to finances, going up two levels (into classes and then into home), back down into documents, and finally down into finances. The call to os.listdir confirms that finances contains just the file money.csv. Finally, we create the variable old_path that refers to the path to the current location of money.csv. We can now move the file.

```
>>> new_path = 'money.csv'
>>> os.replace(old_path, new_path)
```

We then create the path where we want money.csv to move. Since we want to move it into the current working directory, the relative path is just the name of the file. Finally, we call os.replace to complete the move. We can now use os.listdir to confirm that the move was successful. The finances directory is now empty.

```
>>> os.listdir(finances)
[]
```

The programming directory has gained a new file, `money.csv`.

```
>>> os.listdir()
[' bill .txt ', ' billing .py', 'hello_world.py', 'money.csv']
```

A word of warning. If you use os.replace to move a file to a location where another file already exists, the existing file will be overwritten and permanently deleted. It is a good practice to use os.path.exists to ensure that the file does not already exist before calling os.replace.

Copying files. What if, rather than moving money.csv from the finances directory to the programming directory, we had simply wanted to make a copy instead? We can do so using the shutil.copyfile function, which works identically to the os.replace function except that it creates a copy rather than removing the file from its old location. Just like os.replace, it takes two string arguments: the path to the file's current location and the path where the copy should be stored.

```
>>> import shutil
>>> os.listdir(finances)
[' money.csv']
>>> shutil.copyfile(old_path, new_path)
>>> os.listdir(finances)
[' money.csv']
>>> os.listdir()
[' bill .txt ', ' billing .py', 'hello_world.py', 'money.csv']
```

The first command imports the shutil library, which we need to do before we can use it. The second command shows that the finances directory contains the file money.csv. We then call the shutil.copyfile command, asking it to copy this file into the programming directory. Other than the function name, the command is exactly the same as the one we used to move the file. Finally, we check to ensure that finances still contains memo.pdf and that a copy was successfully made in programming.

A word of warning. Just like os.replace, if the new path of shutil.copyfile refers to a file that already exists, the existing file will be overwritten and permanently deleted.

Deleting files. The final file-related operation we will discuss is os.remove, which takes as its argument a string containing a path to a file and, unsurprisingly, deletes that file. No word of warning is necessary to convey the danger of this operation: any files you delete with os.remove are lost forever.

Once we are done using money.csv in the programming directory, we can use os.remove to delete it.

```
>>> os.listdir()
[' bill.txt', 'billing.py', 'hello_world.py', 'money.csv']
>>> os.remove('money.csv')
>>> os.listdir()
[' bill.txt', 'billing.py', 'hello_world.py']
```

This function is rather straightforward. If we wanted to delete a file elsewhere on the computer, we need only provide the appropriate path as an argument.

12.4.2 Manipulating Directories

Creating directories. The os library includes the os.makedirs function, which allows you to create new directories. It takes as its sole argument a string path to a directory to be created. Python then creates that directory.

Suppose we wanted to create a directory called conlaw inside the classes directory. Again, our current working directory is the programming directory.

```
>>> conlaw = os.path.join('..', 'conlaw')
>>> os.path.exists(conlaw)
False
```

First, we create a path to the directory that we want to create. From the programming directory, we go up one level ('..') and then down into the new directory ('conlaw'). Using os.path.exists, we confirm that the directory doesn't yet exist. We can now use the os.makedirs function to create the directory.

```
>>> os.makedirs(conlaw)
>>> os.path.isdir(conlaw)
True
```

The os.path.isdir function confirms that the directory was successfully created.

The os.makedirs function will create multiple levels of directories if they don't already exist (hence the name "makedirs"). For example, what if we wanted to create a new series of directories, one inside the other, within home to store videos. Within home, we will create a new directory called videos, which will have a child directory called tv, which will have a child directory called law_and_order.

One way of creating these directories would be to call os.makedirs three times. First, we create /home/videos.

```
>>> videos = os.path.join('..', '..', 'videos')
>>> os.makedirs(videos)
```

We then create the child directory.

```
>>> os.makedirs(os.path.join(videos, 'tv'))
```

Finally, we create the grandchild directory.

```
>>> os.makedirs(os.path.join(videos, 'tv', 'law_and_order'))
```

We can be more efficient and compress these three calls to os.makedirs into one. If os.makedirs is provided with a path that ends with several new directories (such as /home/videos/tv/law_and_order, the last three of which are new), it will create all of those directories one after the other. In other words, we could have achieved the same effect by simply calling

os.makedirs(os.path.join(videos, tv, law_and_order))

at the very start.

Moving and renaming directories. The os.replace function works the same way for dictionaries as it does for files. It takes two arguments: a path where the directory is currently located and the path where it should move. For example, we could rename the pictures directory to photos with the following function call:

```
>>> documents = os.path.join('..', '..', 'documents')
>>> pictures = os.path.join(documents, 'pictures')
>>> photos = os.path.join(documents, 'photos')
>>> os.replace(pictures, photos)
```

We could also use the os.replace function to move the newly-renamed photos directory up one level from a child of documents to a child of home (and a sibling of documents).

```
>>> new_path = os.path.join('..', '..', 'photos')
>>> os.replace(photos, new_path)
```

There is one key difference between using os.replace on files and on directories. If the new name refers to a file that already exists, the existing file will be overwritten and permanently deleted. If the new name refers to a *directory* that already exists, the program will crash with an error message.

Copying directories. The shutil library provides a function called shutil.copytree that copies a directory from one location to another without removing it from its current location. Its arguments take the same form as those to os.replace: a path referring to the directory's current location and a path referring to the location to which the directory should be copied. This function will copy

the directory and all of its descendant files and directories. Just as with os.replace, if the location to which the directory is to be copied is a directory that already exists, the program will crash..

Deleting directories. The shutil.rmtree function permanently removes an directory and all of its descendants. It takes a single string argument: a path to the directory to be deleted. Use this function with extreme caution—a single careless call to shutil.rmtree could delete every file on your computer.

As a safer but less convenient alternative to shutil.rmtree, the os.rmdir function also removes the directory specified by its string argument. The difference is that os.rmdir will only delete the directory if it is empty. In order to use os.rmdir, you will need to individually delete each of the directory's child files and directories. Although doing this extra deletion is tedious, this inconvenience is often a reasonable price to pay for the safety that comes with os.rmdir. If you accidentally use os.rmdir on the root of your filesystem, it will fail because the root is not an empty directory; if you do the same with shutil.rmtree, it will willingly delete everything on your computer.

Other functions. The functions presented in this section are only a small portion of the wide variety of file and directory manipulation functions that the os and shutil libraries make available. Although these functions are sufficient to perform all of the basic filesystem manipulation tasks, you may find other functions particularly convenient for certain situations. If you want to explore these libraries further, we encourage you to look at the official Python documentation:

- `https://docs.python.org/3/library/os.html`
- `https://docs.python.org/3/library/shutil.html`

12.5 Summary

In this chapter, you learned how to interact with the permanent storage available on your computer's hard drive. Although you had already worked with text files in previous chapters, we reviewed that content and fit it into the context of the far larger universe of binary files. After this warmup, we discussed the way files and directories are organized on your computer's filesystem. You learned how to construct paths between different locations in the file tree, making it possible for your programs to access any file on your hard drive. Finally, we introduced the os and shutil libraries for exploring the filesystem and manipulating its structure. In summary, you now have the ability to explore, interact with, and rearrange the structure of your computer's filesystem, making it possible for your programs to create, organize, and retrieve data intended for long-term storage.

This chapter concludes Part I of this book. Over the preceding thirteen chapters, you have learned what it means to program (Chapter 2), how to store

data (Chapters 5, 8, 9, 11, and 13), and how to manipulate it (Chapters 6, 7, 10, and 12). The tools you have learned along the way—variables, conditionals, loops, lists, dictionaries, and files—are the building blocks of every program you will write in Python or any other language. Although these concepts appear under different names in slightly different forms, they are fundamental ideas common to every modern programming language. If you actively engaged with and internalized the preceding thirteen chapters, you have all the knowledge you need to call yourself a proficient programmer. In the second half of this book, we will put those skills to work on concrete applications that will be valuable in your day-to-day life as a lawyer.

12.6 Cheatsheet

Reading from and Writing to Files

A file is a piece of data stored on your computer's hard drive. Files have names to distinguish them from one another and to give you a way to refer to them. A name (e.g., hello.py) has two parts, the name before the dot (e.g., hello) and the extension after the dot (e.g., py). The extension indicates how a program should interpret the 0's and 1's underlying the file (e.g., as an image, as a Python file, as a Word document, etc.).

Text files Text files are files where the underlying 0's and 1's represent letters, numbers, and other characters. They can be treated as files containing strings.

Writing to a text file.

1. Open the file in write-mode, saving the resulting filehandle to a variable. When opening a file in write-mode, it will be created if it doesn't already exist and overwritten with an empty file if it does.

   >>> fh = open('bill.txt', 'w')

2. Write one or more strings to the file. You can call the write method more than once, concatenating the argument onto the end of the file.

   >>> fh.write('$1,542')

3. Close the file.

   >>> fh.close()

Reading from a text file.

1. Open the file in read-mode.

   >>> fh = open('bill.txt', 'r')

2. Read the contents of the file into a string.

   >>> contents = fh.read()
   >>> contents
   '$1,542'

3. Close the file.

   >>> fh.close()

Appending to an existing file. To write to the end of an existing file, open the file in append-mode by passing open the string `'a'` as its second argument. Doing so will open the file for writing without deleting the existing version of the file if it already exists.

Binary files. Binary files include all files, even those whose underlying binary represents something other than text (e.g., images, PDF documents). Without the help of specialized libraries, you cannot make sense of the contents of these files as you would the string contents of a text file. However, you can read the contents of a binary file into a byte string and write that byte string back to another binary file.

Copying from one binary file to another.

1. Open the file in binary read-mode. Note that reading in binary mode requires using the second argument `'rb'`.

   >>> fh = open('picture.jpg', 'rb')

2. Read the contents of the file into a byte string.

   >>> contents = fh.read()

3. Close the file.

   >>> fh.close()

4. Open the copy in binary write-mode. The second argument must be `'wb'` for binary write-mode.

   >>> fh_copy = open('picture_copy.jpg', 'wb')

5. Write the byte string to the file.

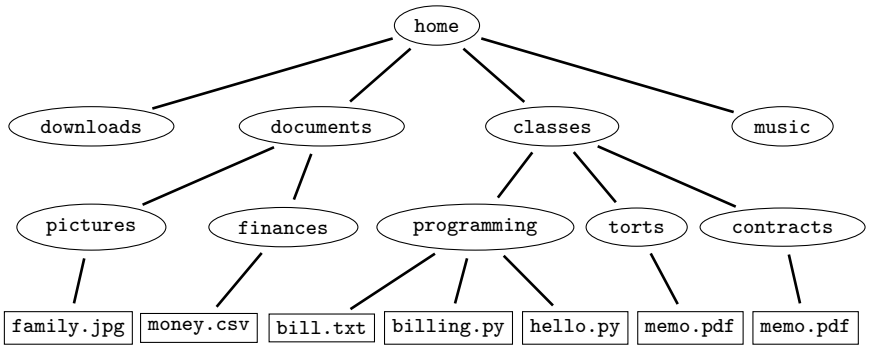
   >>> fh_copy.write(contents)

6. Close the file.

   >>> fh.close()

The Filesystem

The files on your computer are collected into various "locations" known as directories. A directory can store both files and other directories which may, themselves, store other files and directories. Every directory is stored within a higher level directory all the way up to the root of the filesystem. An example filesystem is below. Each oval is a directory and each rectangle is a file.



The items stored inside a directory are its children. The `documents` directory has two children: `pictures` and `finances`. The directory in which an item is stored is called its parent. The `documents` directory is the parent of `pictures` and `finances`. The `home` directory is implicitly a child of the root directory, which is at the top of the filesystem.

Paths. A path describes a way of getting from one location on the filesystem to another. For example, the path from the root to the file `bill.txt` involves entering the `home` directory, then entering the `classes` directory, then entering the `programming` directory, then arriving at the `bill.txt` file. A path that starts at the root is known as an absolute path.

In contrast, a relative path is a path that describes getting from one non-root location on the filesystem to another. For example, a relative path to get from inside the `programming` directory to the file `money.csv` involves going up one level to the `classes` directory, up another level to the `home` directory, down into the `documents` directory, down into the `finances` directory, and then to `money.csv`.

Writing paths. Paths look different on MacOS and Linux as compared to Windows. On MacOS and Linux, a path is written as a string where each step along the way is separated by a forward slash (/). Starting at the root is signified by beginning the path with an initial forward slash. The absolute path from `home` to `bill.txt` is written as:

/home/classes/programming/bill.txt

On Windows, steps in the path are separated by backslashes (\) and the root directory is written using the letter of the particular hard drive in use.

C:\home\classes\programming\bill.txt

Relative paths look slightly different than absolute paths. Since they don't begin at the root, they don't include the starting / or C:\. Relative paths express the notion of taking a step up one level in the filesystem with two dots (..). For example, the relative path from inside `programming` to `money.csv` in MacOS and Linux is written as below:

../../ documents/finances/money.csv

The first .. moves from inside `programming` to inside `classes`, and the second moves from there to inside `home`. The path then moves down into `documents`, `finances`, and finally reaches `money.csv`.

On Windows, this path looks identical with backslashes replacing forward slashes.

..\..\ documents\finances\money.csv

Building paths in Python. To ensure that your programs can work on any operating system, you should never write forward or backwards slashes into paths. Instead, use the os.path.join function provided by importing the os library. The os.path.join function joins all of its arguments together using the appropriate path separator (accessible as os.sep). For example,

os.path.join ('..', '..', 'documents', 'finances', 'money.csv')

evaluates to

../../ documents/finances/money.csv

on MacOS and Linux and

..\..\ documents\finances\money.csv

on Windows.

To create an absolute path with os.path.join, include os.sep as the first argument.

os.path.join(os.sep, 'home', 'classes', 'programming', 'bill.txt')

evaluates to

/home/classes/programming/bill.txt

on MacOS and Linux and

C:\home\classes\programming\bill.txt

on Windows.

The current working directory. When you run a Python program, all relative paths are calculated relative to the current working directory. When your program first starts, the current working directory is the directory where the shell had navigated to before executing Python (typically the directory where the program is stored). If you want to open a file somewhere else on your filesystem, you can do so by providing the appropriate relative path from the current working directory.

You can access the current working directory using the os.getcwd function, which takes no arguments and returns a string containing the absolute path to the current working directory.

You can change the current working directory by providing a string path from the current working directory to the new current working directory as the argument to a call to the os.chdir function.

Exploring the Filesystem

- The os.path.exists function takes a path as an argument and returns True if the file or directory referred to by the path exists and False otherwise.
- The os.path.isfile function takes a path as an argument and returns True if the path refers to a file that exists and False otherwise.
- The os.path.isdir function is identical to os.path.isfile but for directories.
- The os.listdir function returns the list of names of child files and directories stored in a particular directory. If no argument is provided, it returns this list for the current working directory. If an argument path to a directory is provided, it returns this list for that directory. This function returns only the names of the items (for example, ['pictures', 'finances']), *not* the full paths.

File and Directory Management

Some of these functions rely in the os library and others the shutil library. Be sure to import any libraries before you use them.

Be very careful when using these functions. When files get deleted by these functions, they are permanently deleted forever. You can cause permanent damage to your computer if you are not careful.

- *Renaming and moving files.* The os.replace takes two arguments: a string path to the location of a file and a string path to the new location where the file should be stored. Renaming and moving files are identical—renaming is just moving a file to a new location with a new name. **A word of warning:** if a file already exists at the new location, it will be overwritten and permanently deleted.
- *Copying files.* The shutil.copyfile takes arguments identical to os.replace. Rather than moving a file from the first location to the second, it copies the file from the first location to the second.
- *Deleting files.* The os.remove function takes as its argument a path to a file. It deletes that file. **Please exercise extreme caution when using this function.**
- *Creating directories.* The os.makedirs takes a string argument containing a path to a directory. It creates all dictionaries at the end of the path that have not yet been created.

- *Renaming and moving directories.* The os.replace takes identical arguments and behaves nearly identically for directories as for files. The only difference is that, if the new directory path already exists as a directory, the call to os.replace will fail and Python will crash.

- *Copying directories.* The shutil.copytree works identically to shutil.copyfile except that it works on directories rather than files.

- *Deleting directories.* The shutil.rmtree takes as its sole argument a path containing a directory to be deleted. This function will delete the directory and all of its descendants irreversibly. **Please exercise extraordinary caution when using this function.** As an alternative, the os.rmdir function takes an argument of the same form but will only delete the directory if it is empty. This approach, while more tedious, is much safer and can prevent you from accidentally deleting files.

Keywords

Absolute path—A set of directions to get from the root of the filesystem to another location in the filesystem.

Ancestor—A directory accessible by moving upwards through the filesystem hierarchy from a particular starting location.

Binary—A sequence of 0's and 1's. At a very low level, all digital data is somehow written as 0's and 1's.

Byte—A collection of eight bits. Typically the smallest unit of data storage on a modern digital computer.

Byte string—A special Python type similar to a string but designed to hold bytes rather than human-readable characters.

Child—A directory is a child of another directory if it is stored directly within that directory.

Current working directory—The directory from which a Python program computes all of its relative paths.

Descendant—A file or directory is a descendant of another directory if you can reach the other directory by moving upwards from the file or directory.

Extension—The part of a filename after the dot. It tells other programs how to interpret the 0's and 1's underlying the file. Examples: `txt` for text files, `pdf` for PDF files, `jpg` for picture files.

Directory—A location in a filesystem that stores files and other directories.

File—A piece of information that is stored indefinitely on your computer's hard drive.

Filehandle—A special Python type representing a communication channel between your program and your computer's operating system. With a filehandle, you can send commands for reading or writing the file. The `open` function returns a filehandle.

Hard drive—The physical piece of hardware in your computer that saves data for long-term storage.

Parent—One directory is the parent of another when it stores that directory.

Platform-independence—The aspiration that a program should run equally well no matter whether it is running on MacOS, Linux, Windows, or any other operating system with Python support.

Recursive—Repeating the same structure at every level of the hierarchy. Example: a filesystem is recursive because directories can store other directories.

Relative path—A set of directions to get from one location on your filesystem to another.

Root—The directory at the very top of any filesystem.

Part II

Applications of Python

Chapter 13

Regular Expressions

The lawyer's stock-and-trade is text. Cases are texts. So are contracts, regulations, statutes, complaints, tender offer letters, and interrogatories. You get the picture. To practice law is to read, organize, and write texts. A good lawyer-programmer has a huge arsenal of tools for processing text.

In Chapter 5, we introduced you to strings, the basic datatype used for storing text in Python, and in Chapter 9 we dove deeply into many functions and methods that operate on strings. In this chapter, we introduce you to a far more powerful toolkit for searching and manipulating strings: regular expressions.

If the functions and methods of Chapters 5 and 9 are like hand tools for managing strings, then regular expressions (also called "regex" or "regexp") are a fully-kitted makerspace, with 3D printers and powertools.

Cory Doctorow, the science-fiction author, chimed in on the raging, never-ending debate that asks, "Should everybody learn to code?" His answer was that while not everybody needed to learn to code, everybody needed to learn how to use regular expressions. He compared this knowledge to "typing or spelling." [1]

The good news is that you'll be heeding Doctorow's advice and learning regular expressions in this chapter. The bad news is that to learn to use regular expressions you basically need to learn half another programming language. You use regular expressions from within Python, but regular expressions have their own syntax and rules that are nothing like the syntax and rules of Python. Just when you thought you were beginning to learn Python, we're going to force you to start from scratch!

[1] https://www.theguardian.com/technology/2012/dec/04/ict-teach-kids-regular-expressions

13.1 Using Regular Expressions in Python

13.1.1 Doing things the hard way with strings

Anything you can do with regular expressions you could do with strings and a lot of Python code instead. For example, in chapter 9, we introduced you to the .find() and .index(), and .count() methods. Recall:

```
1 >>> text = "All legislative Powers herein granted shall be vested in a
      Congress of the United States, which shall consist of a Senate and
      House of Representatives."
2 >>> text.find('e')
3 5
4 >>> text.index('e')
5 5
6 >>> text.count('e')
7 20
8 >>>
```

What if you wanted to use these functions to find not just the first 'e' in every string (as in lines 2 and 4) but instead to return the location of *every* 'e' in the string? What if you wanted to find every vowel? What if you wanted to find every vowel followed by a consonant?

You could accomplish any of these tasks with these functions and if-statements and loops, but many of them would require a lot of code and a lot of messy, error-prone bookkeeping.

Regular expressions allow you to accomplish tasks like these much more efficiently.

13.1.2 Your First Regular Expression

You can think of a regular expression as a glorified search query–the string of text you type into the Google search bar or Westlaw search box. Just as you do when you use these services, you can create small strings–called *regular expressions* or *patterns*–that you can use to search through a long text for matches.

Your first regular expression is just a string of characters you'd like to find in a text. Let's work in this chapter with the text of the U.S. Constitution. What if you wanted to search for the word "state" in the constitution? The regular expression for this search would simply be state.

We'll make this far more complex in a minute. But first, let's look at the library bits–the glue–that allows you to use regular expressions like this one in Python.

Because regular expressions enable you to search through texts, let's first load the U.S. Constitution into a Python string. You can download the copy of the full text we will be working on from https://www.usconstitution.net/const.txt. Just save a copy of this page from your browser into the directory where you

place your Python code. Then, simply use the following lines at the start of your code for the rest of the exercises below:

```
1 import re
2
3 text = open('const.txt').read()
```

13.1.3 The re library and re.search() function

The text variable now includes the full text of the Constitution stored as a very long string. How do you use regular expressions to search for the word "state" in this string? Python provides a library called re that can search through texts using regular expressions. The basic search function to use is called re.search().

```
7 match = re.search('state', text)
8 print(match)
```

In line 7, we call the search function from the re library, which we imported back in line 2. This function takes at least two arguments. The first argument is the pattern, which is simply a string written according to the rules of regular expressions.[2] The second argument is the text to be searched.

13.1.4 Match objects

When the regular expression matches characters in the text to be searched, the re.search() function returns what is known as a "match object". Printing the match object, as in line 8, reveals that a match object was returned. The output of the code above reports something like:

```
% python constitution.py
<_sre.SRE_Match object; span=(17056, 17061), match='state'>
```

To view the part of the text of the Constitution that matched the pattern, you have to use this slightly ungainly form (replacing the previous code snippet):

```
7 match = re.search('state', text)
8 print(match.group(0))
```

We'll explain why the method is called .group() much later. This produces the following:

```
% python constitution.py
state
```

[2] If you do any research on regular expressions in Python, you're sure to run into "compiled" regular expressions. Python allows you to pre-cook your regular expressions into an object that you can then use to conduct your search() functions, yielding potential efficiency and readability benefits. Especially with more recent versions of Python, this isn't strictly necessary, so we are not explicitly compiling the regular expressions in this book.

It's a little underwhelming, don't you think? This output reveals that the search pattern 'state' produced at least one match in the text of the Constitution. Let's break down a bit what is happening here.

First, re.search() (and all of the other methods we will be reviewing in this chapter) begins scanning the string stored in the text variable from front to end. Then, at each location in the file, it searches for a "match" to the pattern. In this case, it searches for the five lower-cased letters, s-t-a-t-e, in that order, without any spaces or other breaks.

When re.search() finds the first match, or hit, it returns a match object. The .group(0) method call returns the text that triggered the match, in this case, the string "state", unsurprisingly.

To produce slightly more useful output, you can use the .start() and .end() methods of the match object like so:

```
7 match = re.search('state', text)
8 print(match.start())
9 print(match.end())
```

This produces the following:

```
% python constitution.py
17056
17061
```

This means that the re.search() function found a match beginning at character 17,056 of the Constitution and ending just before character 17,061 of the Constitution. Just like slice indexing, the start represents the first index of the text string that matched while the end represents one more than the final index of the text string. You could run print(text[17056:17061]) to confirm that the substring at this position is "state".

You could also use a well-positioned slice to discover the context around this first use of "state" in the Constitution. Say, for example, that you want to print the matched letters "state" surrounded by the twenty characters preceding and the twenty characters following the first match. The slice to extract this would be print(text[17056 − 20:17061 + 20]) or simply print (text [17036:17081]) which would return: ``President shall, at stated Times, receive for ''.

Interestingly, notice that this matched the word "stated", which might not be what you intended! We'll teach you some regex tricks that can avoid this "false positive" later in the chapter.

Match objects also provide .span() methods that combine both .start() and .end() in one, returning a tuple that returns both start and end.

```
7 match = re.search('state', text)
8 print(match.span())
```

This produces the following:

```
% python constitution.py
(17056, 17061)
```

What happens if you search for a pattern that doesn't match any of the text? What if you try, for example, the following:

```
7  match = re.search('privacy', text)
8  print(match)
```

Because the word 'privacy' does not occur in the Constitution, this returns the following:

```
% python constitution.py
None
```

And because the Python value None evaluates to a boolean False, it's typical to see code like this:

```
7  match = re.search('privacy', text)
8
9  if match:
10     print(match.group(0))
11     # More processing code here.
12 else:
13     print ('No match found for pattern privacy.')
```

13.1.5 Matching more than just the first hit

You probably want to match more than the first search 'hit' for your pattern. You can use slices together with match.end() to find the second (and subsequent) search 'hits':

```
7  match = re.search('state', text)
8  print ('Matched at position start=' + str(match.start()) + ' and end=' +
       str(match.end()))
9
10 match = re.search('state', text[match.end():])
11 print ('Matched at position start=' + str(match.start()) + ' and end=' +
       str(match.end()))
```

This returns

```
% python constitution.py
Matched at position start=17056 and end=17061
Matched at position start=2979 and end=2984
```

Notice that the second hit reports a position starting at 2979, but that's in the context of the slice. To convert this position to a position corresponding to the original text string, you need to add it to the position of the start of the slice, which was 17061. So 17061 + 2979 = 20040.

In this manner, it would be possible to build a while-loop that kept sending successively smaller/later slices to re.search(), eventually returning all of the hits. We leave this as an exercise for the reader.

Python makes it easy to return *all* of the hits for a given pattern in a given text, with the re.finditer() function. It has the same syntax as re.search() but it returns an object that itself returns each match object, one at a time. Like the range() function, you can send re.finditer()'s output to list() or use it in a for-loop to process the match objects one-by-one:

```
7 for match in re.finditer('state', text):
8     print ('Matched at position start=' + str(match.start()) + ' and end='
          + str(match.end()))
```

Produces the following:

```
% python constitution.py
Matched at position start=17056 and end=17061
Matched at position start=20040 and end=20045
Matched at position start=30176 and end=30181
Matched at position start=30299 and end=30304
Matched at position start=31375 and end=31380
Matched at position start=31412 and end=31417
Matched at position start=31521 and end=31526
Matched at position start=31555 and end=31560
```

A simpler function called re.findall() returns just the string hits without all of the other information found in a match object (like start and end positions).

```
7 matches = re.findall('state', text)
8 print (len(matches))
9 print (matches)
```

Produces the following:

```
% python constitution.py
8
['state', 'state', 'state', 'state', 'state', 'state', 'state', 'state']
```

With a simple pattern like the one we are using, this output isn't terribly illuminating, but you'll soon see why this is very powerful:

13.1.6 The dual purposes of a regular expression

Every regular expression, when given action through a Python function, fulfills two different purposes.

First, a regular expression acts a little like a boolean test. The pattern is compared to every single position in the text string to answer the repeated question, "Is there a hit at this spot?"

To be clear, the reason we say that this acts just "a *little* like a boolean test" is that that these Python functions you're learning do not return a True or False value. Instead, every regular expression marches down the text searched, asking at every position, "Is this a match?" If the answer is no, the search moves to the next position. If the answer is yes, something happens, depending on the function. But a literal True or False is not returned.

Second, if there is a hit, the regular expression directs what part of the text is returned to the user. This is the string that gets packaged into the match object (in the cases of of re.search() and re. finditer ()) or the string that gets added to the list to be returned (in the case of re. findall ()).

To this point, with the trivially simple pattern we have been using, there may not seem to be much difference between these two purposes. The two seem inextricably intertwined. To fulfill the first purpose, the regular expression searches for particular characters, and to fulfill the second purpose, it returns those very same characters.

We emphasize the dual purposes, however, because later patterns will put some daylight between these two purposes. In particular, some patterns will have components that contribute to its *boolean purpose* but that will not add anything to the *return value*. To better understand those examples, it's useful to emphasize the dual purposes at this simpler moment in the description.

13.2 Intermediate regex: More powerful patterns

So far, regular expressions don't improve much on string methods like string .find(). The secret is in the power and variability of what you can pass as a pattern to the re.search(), re. findall (), and re. finditer () functions. This chapter can't capture the full richness of regular expressions, but here are some common pattern enhancements.

13.2.1 Case-insensitive search

A keen student of the Constitution might be surprised to learn that the word "state" appears in the U.S. Constitution only eight times. In reality, it occurs more often than that, but regular expression searches are by default case sensitive. Our federalism-loving founding fathers apparently used "State" much more often than the lowercase version.

All of the re methods discussed so far take an optional third argument known as the "flags" argument. This is a "keyword argument", meaning you must precede it with the string "flags=". By the end of the chapter, you'll have learned several possible flags for Python regular expressions. The first is the ungainly constant value re.IGNORECASE. When passed to one of the re functions,

it signifies that the search should ignore the case of each letter in the pattern during the search. So the following:

```
7 matches = re.findall('state', text, flags=re.IGNORECASE):
8 print (len(matches))
```

Returns:

```
% python constitution.py
211
```

The Constitution (including all of the amendments) use the word "state" or some variant of it, 211 times!

13.2.2 The . special character

Up to this point, our regular expression patterns have found only strings that matched on an exact character-by-character basis. The pattern 'state' matched only those five characters occurring in that order.

Note that simple patterns like this first one can match more than alphabetic characters. They can also contain numerals, spaces and punctuation marks, all of which will make simple matches. So "United States" and "Hello, World!" and "1337 hax0r" are all valid patterns that can be sent to any of the re functions.

Patterns can also contain many different *special characters* that are much more flexible than literal characters. The first, and perhaps most fundamental, one is the dot character, just a single . Each . character matches one and only one character in its place in the pattern. "Character" is meant broadly here: the . will match spaces and tabs and numbers and letters and punctuation marks. It is a universal match placeholder, also called a wildcard. The only thing a . will not match, at least not by default, is a newline character.

For example, imagine you wanted to find all mentions in the Constitution of the person second-in-rank in the Executive Branch. Interestingly, the Constitution refers to this person in two ways: as the hyphenated "Vice-President" and the unhyphenated "Vice President". With the . character, you can capture both in a single pattern with code like the following:

```
7 matches = re.findall('vice.president', text, flags=re.IGNORECASE)
8 print (matches)
9 print (len(matches))
```

Note the dot character between "vice" and "president" in the pattern on line 6. This returns:

```
% python constitution.py
['Vice President', 'Vice President', 'Vice−President', 'Vice President', '
    Vice−President', 'Vice President', 'Vice President', 'Vice President', '
    Vice−President', 'Vice−President', 'Vice−President', 'Vice−President', '
    Vice−President', 'Vice−President', 'Vice−President', 'Vice−President', '
```

```
 Vice−President', 'Vice−President', 'Vice President', 'Vice President', '
 Vice President', 'Vice President', 'Vice President', 'Vice President', '
 Vice President', 'Vice President', 'Vice President', 'Vice President', '
 Vice President', 'Vice President', 'Vice President', 'Vice President', '
 Vice President', 'Vice President', 'Vice President', 'Vice President']
36
```

Note further that because the dot character matches *any* character other than newline, this would also capture "Vice_President" or "ViceyPresident" or "Vice!President" (none of which actually appear in the Constitution).

Also, because dot doesn't match the newline character, if our text-file version of the Constitution happened to split a line between "Vice" and "President", this would not have been matched by this pattern. This doesn't happen to be true of the version of the Constitution we found online.

Both of these issues will be addressed later in the chapter.

13.2.3 Matching specific characters with []

Sometimes, a dot is *too* flexible. Like in the ViceyPresident example, the dot special character will match everything it encounters in its place, even if you know that the true possibilities at that position are limited.

If you know that a given position in the pattern can match only one of a finite group of possible characters, which we call a character set, you can use square brackets to contain all of the possibilities.

For example (admittedly, not the best one in this chapter), what if you wanted to match both the words "gold" and "good" in the Constitution? You could do so with a single pattern:

```
7 matches = re.findall('go[ol]d', text)
8 print (len(matches))
9 print (matches)
```

The key is the pair of square brackets surrounding the letters 'o' and 'l': go[ol]d. When the first square bracket is encountered, it signifies that the next single character to be matched can be "one of several", specifically either 'o' or 'l'.

The search produces the following:

```
% python constitution.py
2
['gold', 'good']
```

Within square brackets, you are not limited to only two possibilities. For example, you could construct a search for any vowel with [aeiou].

Within a character set, you can also use a hyphen between two characters to specify a range of characters. For example, [a−z] will match any single lowercase character, while [3−8] will match any one of the digits 3, 4, 5, 6, 7, or 8.

You can mix these two types of input, so [aeiou3−8] would match any vowel or any number from 3 to 8.

You can also add a caret character '^' immediately following the first square bracket to signify that the set matches anything *not* in the set. In other words, it specifies the inverse or complement of the character set. So [^aeiou] matches any character excluding vowels (which would include numerals, punctuation marks, and spaces), while [^0-9] matches anything excluding numerals.

Notice how character sets possess one feature not yet seen before in this chapter: this is the first time there isn't a one-to-one correspondence between the number of characters in a pattern and the number of characters in matched strings. Even the dot matches one-and-only-one character. A character set by itself matches only a single character, even if the set is much longer.

13.2.4 Character classes

Somec haracter sets are so commonly used that Python supplies shortcuts for them. They take the form backslash-letter such as '\d' or '\s'. We call them "character classes".

There are more character classes than you will ever need to use. You can find a complete list in the documentation for the re library. Some very useful character classes include:

\s	Matches any whitespace character (spaces, tabs, etc)
\S	Matches any non-whitespace character
\d	Matches any numeric character
\D	Matches any non-numeric character
\w	Matches any alphanumeric character
\W	Matches any non-alphanumeric character

Notice that capitalization matters with character classes and that all three examples listed above (s, d, and w) come with capitalized and uncapitalized forms, which mean the opposite of one another.

Before we demonstrate character classes in Python code, recall from the strings chapter the concept of a "raw string". A raw string is a string with the initial quote (single or double) preceded by a lower-case 'r'. Such as r'Test'.

You're finally ready to appreciate why this convention is so useful. What 'r' signifies is that none of the escape sequences, such as '\n', should be interpreted by Python. This is tailor-made for the many backslashes in regexs from character classes. For the rest of the examples in this chapter, we will precede every pattern with an r, and you should too.

Onto the character class examples. '\d' matches a single digit. So the following code:

```
7 matches = re.findall(r'Amendment \d', text)
8 print (len(matches))
9 print (matches)
```

returns:

```
% python constitution.py
27
[' Amendment 1', 'Amendment 2', 'Amendment 3', 'Amendment 4', '
    Amendment 5', 'Amendment 6', 'Amendment 7', 'Amendment 8', '
    Amendment 9', 'Amendment 1', 'Amendment 1', 'Amendment 1', '
    Amendment 1', 'Amendment 1', 'Amendment 1', 'Amendment 1', '
    Amendment 1', 'Amendment 1', 'Amendment 1', 'Amendment 2', '
    Amendment 2', 'Amendment 2', 'Amendment 2', 'Amendment 2', '
    Amendment 2', 'Amendment 2', 'Amendment 2']
```

Notice that this might not be what you intended. The character class '\d' matched the '1' in 10, 11, 12, etc, and the '2' in 20, 21, 21, etc. If you wanted just the single-digit Amendments, you could use the '\s' character class instead:

```
7 matches = re.findall(r'Amendment \d\s', text)
8 print (matches)
```

This outputs:

```
% python constitution.py
[' Amendment 1\n', 'Amendment 2\n', 'Amendment 3\n', 'Amendment 4\n', '
    Amendment 5\n', 'Amendment 6\n', 'Amendment 7\n', 'Amendment 8\n
    ', 'Amendment 9\n']
```

Notice that the '\s' character class matched (and thus returned) the newline characters ('\n') within the file. A newline character counts as a space in the '\s' character class, so this properly matches 'Amendment 2' but not 'Amendment 20'.

Finally, a '\w' matches any letter or number but not punctuation or spaces. The 'w' stands for 'word.' Conversely, a '\W' matches any non-letter or number.[3]

13.3 Repetition with regex

So far, the power of the regular expressions we have matched has been fairly constrained. We have learned only regex features that match a single corresponding character.

To craft regular expressions that are far more flexible and powerful, we need to introduce a few ways to signify repetition. These conventions allow you to say that a particular character appears "zero or more" times or "zero or one" times or "one or more" times.

[3] '\w' also matches the underscore ('_') character. But because you won't run into those too often, we relegated this to a footnote!

Unfortunately, this great power comes at a cost to readability. With the addition of repetition, you can no longer read a regular expression from left-to-right and understand exactly how many characters will be matched. In fact, regular expressions are notoriously difficult to read and decipher, in large part because of the features introduced in this section.

13.3.1 The '?' special character

What if you want to match every occurrence of variations of the word "Senator", which sometimes occurs as the singular "Senator" and sometimes as the plural "Senators"? The '?' special character can help you do this. Unlike the previous pattern components you have seen, the '?' is a *modifier*, which refers to the character that immediately precedes it to the left. '?' tries to match that character zero or one times.

So to match both "Senator" and "Senators", you would use the pattern Senators?. The '?' modifies only the 's' that immediately precedes it. It leaves unaffected the S-e-n-a-t-o-r that it does not adjoin. This pattern will match if the final 's' is present ("Senators") or if it is absent ("Senator").

In code:

```
7 matches = re.findall('Senators?', text)
8 print (len(matches))
9 print (matches)
```

Returns:

```
% python constitution.py
21
['Senators', 'Senator', 'Senators', 'Senator', 'Senators', 'Senators', '
    Senators', 'Senator', 'Senators', 'Senator', 'Senators', 'Senators', '
    Senators', 'Senator', 'Senators', 'Senator', 'Senator', 'Senators', '
    Senators', 'Senator', 'Senators']
```

13.3.2 The '+' special character

For the remaining repetition features, we will leave our study of the Constitution. James Madison and the other framers wrote pretty concise prose, not well suited to the kind of brainteasers you can construct with the following regex features.

Let's operate on the string 'aaaaaaaabbbbbbbbccccccc' instead. We'll show examples in the Python shell. So, for the following, assume these commands are run first:

```
% python
>>> import re
>>> testString = 'aaaaaaaabbbbbbbbccccccc'
```

The '+' special character is related to the '?' special character. '+' also modifies the character preceding it, but trying to match that character one-or-more times.

Consider the following examples:

```
% python
>>> print(re.findall('a+', testString))
['aaaaaaaa']
>>> print(re.findall('a+b', testString))
['aaaaaaaab']
>>> print(re.findall('a+b+', testString))
['aaaaaaaabbbbbbbb']
>>> print(re.findall('a+bc', testString))
[]
>>> print(re.findall('a+b+c', testString))
['aaaaaaaabbbbbbbbc']
>>> print(re.findall('a+b+c+', testString))
['aaaaaaaabbbbbbbbccccccccc']
```

Study each of these closely to make sure you understand how '+' operates. Why didn't a+bc match anything?

The '+' modifier can modify not only literal characters but also any regex feature listed above.

Consider this example:

```
% python
>>> print(re.findall('.+', testString))
['aaaaaaaabbbbbbbbccccccccc']
```

What happened here? Recall that `.` matches every character, so the '+' modifier kept matching until it ran the length of the testString. Notice this subtlety: It didn't matter that the characters in the string changed from 'a' to 'b' to 'c'. The '+' modifier kept looking for the next character that potentially matched `.` , which was always true.

Here are some additional examples of '+':

```
>>> print(re.findall('[ab]+', testString))
['aaaaaaaabbbbbbbb']
>>> print(re.findall('[ac]+', testString))
['aaaaaaaa', 'cccccccc']
>>> print(re.findall('\s+', testString))
[]
>>> print(re.findall('a.+', testString))
['aaaaaaaabbbbbbbbccccccccc']
>>> print(re.findall('b.+', testString))
['bbbbbbbbccccccccc']
```

Again, study each example and ensure you understand what is happening. Why did [ac]+ produce two matches? Why didn't \s+ produce any matches?

The '[ac]+' example illuminates something about the .findall() (and .finditer()) commands. These functions, which are designed to search for more than one regex matches, begin each search where the prior search ended.

In other words, the pattern '[ac]+' matches 'aaaaaaaa' as its first hit because this string of eight consecutive a's is "either a or c" eight times in a row. But after returning this hit as this first match, the pattern begins searching for a second match *immediately after* the end of the first search, so in the spot between the last 'a' and the first 'b'.

This explains why .findall() didn't return, in addition to the 'aaaaaaaa' hit, a second hit of 'aaaaaaa' (seven 'a's) or 'aaa' (three 'a's), matching substrings further down the long string of eight 'a's. After the first match, all of the 'a's had been exhausted by the search, and the next search commenced at the 'b's.

13.3.3 The '*' special character

Perhaps the most frequent special character you will use in constructing regular expressions is the '*' character, which modifies the character preceding it by trying to match it zero or more times. It is like '+' with the crucial difference of matching even when the preceding character is absent. This small difference can lead to significantly different behavior.

Start with some fairly straightforward examples:

```
>>> print(re.findall('aa*', testString))
['aaaaaaaa']
>>> print(re.findall('bb*', testString))
['bbbbbbbb']
>>> print(re.findall('cc*', testString))
['ccccccc']
```

But how do you explain the following?

```
>>> print(re.findall('a*', testString))
['aaaaaaaa', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
    '']
```

Why did re.findall() return a list with eighteen items, the string 'aaaaaaaa' followed by seventeen empty strings? This is the result of the fact that '*' matches 'zero' of the preceding. So translating the pattern 'a*' to English, this tries to match either one of the following:

1. One or more 'a' characters; or
2. The empty string ''. (Because zero a's is just the empty string.)

The first hit to the string 'aaaaaaaa' matches pattern rule 1. It is eight 'a' characters. At this point, the re.findall() function moves to the end of the previous hit, so it analyzes the spot in the string immediately following the last 'a' and immediately before the first 'b'. It first tests rule 1: does this match one or more 'a' characters? No, this test fails. But then because of the way '*' operates, it asks, "Does this spot in the string match the empty string"? Yes,

it does. It matches the empty string between the last 'a' and the first 'b', so it dutifully returns that empty string and proceeds forward.

Let's pause for a moment to remind ourselves about the dual purposes of regular expressions. Any part of any pattern that matches "zero" of something–not just * but also the ? we encountered earlier–serves the boolean purpose of a regular expression without contributing to the return value. So rather than say the pattern "returns that empty string" as did in the prior paragraph, it might be more proper to note that this part of the pattern found a hit (serving the boolean purpose) but did not contribute anything to the return value. Back to the example.

The regular expression then analyzes the first 'b', which fails to match either rule 1 or rule 2. Having failed to make a match, it moves to the spot following the first 'b' but before the second 'b'. Does that spot match the empty string? Sure, so it returns another "hit", which gets added to the two-th spot in the list that's returned.

This continues down the string, returning seventeen empty strings, one for each of the slots between the remaining characters, including one for the slot after the last 'c'.

Yes, this is confusing. There are two morals to the story. First, it's almost never useful or correct to put a single character modified by a '*' alone in a pattern. Second, any time you use the '*' modifier, you have to consider potentially confusing things happening when you match 'zero' times.

13.3.4 Using curly braces for specific number of matches

Finally, if you want to use match repetition with more precision, you can use curly braces to provide specific upper and lower bounds to the number of times the preceding character must occur.

{m,n} means match m to n repetitions. So {2,4} will match if the character preceding matches 2, 3, or 4 times. (Sorry to say it, but unlike Python slices, the upper bound is inclusive rather than exclusive.)

If you want a precise number of matches–meaning the upper and lower bounds are the same–you can just put one number between the curly braces. So {3} means match 3 and exactly 3 times.

Consider the following examples:

```
>>> print(re.findall('a{7}', testString))
['aaaaaaa']
>>> print(re.findall('a{4}', testString))
['aaaa', 'aaaa']
>>> print(re.findall('a{3}', testString))
['aaa', 'aaa']
>>> print(re.findall('ab{3}', testString))
['abbb']
>>> print(re.findall('a{2}b{3}', testString))
['aabbb']
```

```
>>> print(re.findall('a{2}b{8}c{3}', testString))
['aabbbbbbbbccc']
```

Be sure you understand each of these examples. Try to figure out exactly which characters from the testString are being matched in each hit listed above.

13.3.5 Greedy vs. Non-greedy matching

By default, the '+' and the '*' modifier will find as many characters as possible that match the part of the pattern they modify. So, for example, see the interactions in the following snippet of Python code:

```
1 >>> citations = "United States v. Haney, 264 F. 3d 1161, 1164−1166 (CA10
      2001); United States v. Napier, 233 F. 3d 394, 402−404 (CA6 2000);
      Gillespie v. Indianapolis, 185 F. 3d 693, 710−711 (CA7 1999)"
2 >>> first_citation = re.search(".+;", citations )
3 >>> print(first_citation.group(0))
4 United States v. Haney, 264 F. 3d 1161, 1164−1166 (CA10 2001); United
      States v. Napier, 233 F. 3d 394, 402−404 (CA6 2000);
```

Notice that this pattern includes a semicolon that is meant to delineate the end of a citation in a string cite ('.+;'). This pattern didn't stop matching characters when it encountered the first semi-colon but instead kept matching until it found the *last* semi-colon. If this string-cite had 50 citations rather than 3, it would have matched the first 49 of them!

This is called "greedy" matching behavior. The '+' and '*' modifiers are, by default, greedy matchers. They will match any and all characters that match the character they modify (in this case, the dot), stopping only when the part that follows forces it to stop.

This can often lead to surprising results, when a '+' or '*' does not limit itself to the first hit, but instead captures much more than intended. The dot is especially prone to this behavior when modified by '+' or '*', because it matches everything except a newline.

The solution is simple. Both '+' and '*' can be switched from greedy to non-greedy behavior simply by placing a question mark after them. So '+?' and '*?' are the non-greedy versions of these modifiers.

Here is the identical code with a single question mark added to the pattern on line 2.

```
1 >>> citations = "United States v. Haney, 264 F. 3d 1161, 1164−1166 (CA10
      2001); United States v. Napier, 233 F. 3d 394, 402−404 (CA6 2000);
      Gillespie v. Indianapolis, 185 F. 3d 693, 710−711 (CA7 1999)"
2 >>> first_citation = re.search(".+?;", citations )
3 >>> print(first_citation.group(0))
4 United States v. Haney, 264 F. 3d 1161, 1164−1166 (CA10 2001);
```

It returns a single citation, as intended.

For another example, consider this code, which look for a particular bit of information from a snippet of html (the language used for creating web pages):

```
1 html = "<div class='choice'><a href='visit.html'>Click here!</a></div>"
2 print(re.search("<a.+>", html).group(0))
```

produces the following

Click here!

Notice that the dot-plus-bracket ('.+>') part of the pattern didn't stop matching characters with the dot when it first encountered a right angle bracket. Instead, it kept matching more and more characters until it found the last possible right angle bracket.

Once again, the solution is simple.Let's add a single question mark to the code above:

```
1 html = "<div class='choice'><a href='visit.html'>Click here!</a></div>"
2 print(re.search("<a.+?>", html).group(0))
```

This now produces the following

<a href='visit.html'>

Because of the non-greedy '+' modifier, the dot matches only until the very first right angle bracket is found.

13.3.6 Escaping special characters

What if you want to search a text for a period ('.') character? What if you want to search for an asterisk ('*') or an open parenthesis ('(')?

The answer, as we have encountered before in Python, is to use the backslash character to escape the special functionality. Once again, using a raw string (r") is a good idea when dealing with any backslashes.

```
>>> import re
>>> citation = "See Orin S. Kerr, A Theory of Law, 16 Green Bag 2d 111,
    111 (2012)."
>>> year = re.search(r"\((\d+)\)\.", citation)
>>> print (year.group(0))
(2012)
>>> print (year.group(1))
2012
```

13.4 Putting complex regex's together

We've just started to scratch the surface of what regular expressions plus Python's re library can accomplish. Before getting into a few more advanced topics, consider an example that brings together much of what you have already learned: Can you create a set of regular expressions that can identify case citations? Armed with such regular expressions, you can load the text of judicial opinions

or law review articles into Python strings and use re.findall() or re.finditer() to extract all of the citations.

As a refresher, citations obey the following form:

36	F.3d	1303
(Volume)	(Reporter)	(First Page)

For now, we're ignoring party names and the year.

The first part of the citation is the volume. A volume is a multi-digit number. At least for the federal reporters, this starts at Volume 1 and ends at Volume 999. One regular expression snippet that will capture this is: \d{1,3}

Once again, this means "match one, two, or three digits in a row." Now, this isn't a perfect regular expression. For example, the lead digit can never be zero. If this kind of precision is necessary, you can use [1-9]\d{0,2} instead.

The reporter is trickier. The reporter for district court opinions is F. Supp. and F. Supp. 2d. The reporter for appellate court opinions is F., F.2d, and F.3d. The reporter for the Supreme Court is either U.S. or S. Ct. (at least for modern cases).

Eventually, you'll know enough regex tricks to build all of these reporters into a single pattern. For now, let's focus on the appellate court reporters. One strategy you might adopt is to focus on F.2d and F.3d first, given their similarities. A regular expression that can capture both of these is F\.[23]d. Notice the backslash before the period. As with Python, this tells the re module not to interpret the dot as the python wildcard but rather to match only literal periods.

Knowing what you know at this point, can you change this pattern to match the plain, first volume "F." as well? Not obviously. You could try something like F\.[23]?[d]? The problem is that this would match both F.2 and F.d, neither of which is correct. So, for now, you need two separate patterns, one for the F. reporter and one for both F.2d and F.3d. This also means you'll need two separate calls to a function from the re module, one for each pattern.

Finally, what about the page number? Once again, experience teaches that this can be a one to four digit number, without leading zeroes. You can use [1-9]{0,3}.

Even greater experience suggests that the appellate reporters never reach 2000 pages. So for four digit numbers, you might want to include only a leading 1, to the exclusion of all other digits. Once again, you do not know enough regex yet to specify this.

Finally, it's time to stitch these three parts together. As explained earlier, use a raw string beginning with `r'` to surround this pattern. Your only final decision is what to place between the three sub-patterns you've created above. Usually, case citations include a single space between the volume and reporter and a single space between the reporter and page number. But is that universally guaranteed? Might there be some accidentally doubled spaces you might encounter? Is the risk of missing one of those (a false negative) worth the risk that accounting for a doubled space might capture something that isn't a citation (a false positive)?

There is no one right answer to these questions. They involve tradeoffs and depend on your goals and the kind of errors you're most wanting to avoid. One compromise is to use a single \s between each of the three subpatterns. This will cover single spaces but also single newlines, which will capture citations that are broken across the end of a line.

Putting this all together, one possible regular expression pattern for capturing case citations in an opinion is:

```
1 import re
2
3 pattern = r'[1-9]\d{0,2}\sF\.[23]d\s[1-9]\d{0,3}'
4 matches = re.findall(pattern, text)
5 print(matches)
```

To test this regular expression, we downloaded the full text of the D.C. Circuit's 2016 opinion in U.S. Telecom v. FCC, 825 F.3d 674 (2016), a long opinion that takes up 108 pages in the federal reporter and one that is sure to be brimming with citations to other appellate cases. Loading the plain text of this opinion into the text variable and running the code above, produced the following output:

```
['825 F.3d 674', '600 F.3d 642', '740 F.3d 623', '978 F.2d 737', '165 F.3d
    972', '334 F.3d 1096', '653 F.3d 27', '714 F.2d 171', '372 F.3d 441',
    '738 F.3d 397', '450 F.3d 528', '584 F.3d 1076', '742 F.2d 1520', '224
    F.3d 768', '462 F.3d 1', '506 F.3d 1070', '504 F.3d 1318', '533 F.2d
    601', '525 F.2d 630', '198 F.3d 921', '206 F.3d 1', '876 F.2d 994', '763
     F.3d 754', '740 F.3d 692', '533 F.3d 810', '705 F.2d 506', '494 F.3d
    188', '657 F.2d 298', '116 F.3d 520', '27 F.3d 642', '237 F.3d 657',
    '158 F.3d 1364', '462 F.3d 1', '570 F.3d 294', '24 F.3d 1441', '592 F.3
    d 139', '746 F.2d 1492', '452 F.3d 830', '330 F.3d 502', '770 F.3d 961',
     '77 F.3d 504', '108 F.3d 358', '963 F.2d 441', '2 F.3d 453', '589 F.3d
    433', '69 F.3d 600', '587 F.3d 1136', '799 F.3d 1126', '740 F.3d 623',
    '216 F.3d 871', '352 F.3d 415', '730 F.2d 778', '988 F.2d 174', '188 F.3
    d 521', '912 F.2d 463', '359 F.3d 554', '462 F.3d 1', '462 F.3d 1', '825
     F.3d 674', '825 F.3d 674']
```

This returned sixty appellate citations, and a quick glance through the list suggests no false positives. Doubtless there are false negatives, although it is hard to say exactly how many.

Two pieces of final commentary about this example: First, we hope you're impressed by how four lines of code (plus a little manual labor to load the opinion into a string) could spit out this result, which would've been much more laborious if done by hand.

Second, the regular expression we wrote on line 3 is simply horrifying to look at, even more so when you understand that this one is fairly modest compared to some you will write and encounter. You can't avoid the simple fact that regular expressions are ugly and hard to understand. Things will get a bit easier with practice, but there's no avoiding the underlying truth of the matter.

Still, suffering with regular expressions opens the door to all sorts of advanced text handling programs, the kind that will automate a lot of the drudgery of legal practice.

13.5 Groups

Parentheses play a special role in regular expression patterns. They "group" part of a pattern into a separable unit. The parentheses themselves do not match any characters, so with simple regular expressions, they do not change behavior in obvious ways.

But the parentheses enable two powerful new applications: (1) within the pattern itself, they let you treat the surrounded characters as a unit, for modification or alternation; and (2) once the pattern has found a match, they make it easy to refer to subparts of a matched string in the rest of your Python program. Let's take these two applications in turn.

13.5.1 Groups for modifiers and alternation

A grouped set of characters act as a single unit for modifiers. Unlike character sets, such as [aeiou], that mean "one of these", a group of characters, such as (aeiou), mean "all of these, in this order." Any modifier immediately following this group will modify the group as a whole, not the individual characters within the group. So "(aeiou)+" means "match strings that have the characters aeiou, in that order, one or more times". This pattern would match all of the following:

aeiou
aeiouaeiou
aeiouaeiouaeiou

But it wouldn't match any of the following:

aeioaeiuo
uoiea

Groups truly unleash the power of modifiers, allowing much more complex patterns than without groups. For a simple example, we can use groups to address one shortcoming of our earlier case citation regex. Remember that we couldn't match both "F." and "F.2d" with a single pattern. Now, we can:

```
r'F\.([23]d)?'
```

Within the parentheses, you need both of the specified characters–the character set that captures 2 or 3 and the letter d–to count as a match. You can not match this group unless you have both characters, meaning 2d or 3d. Thus, unlike the previous attempt to fix this, this pattern will not match "F.2" or "F.d".

In addition to modification, groups also enable alternation of sequences of characters. Remember that square brackets specify a character set–a way of

saying "one of these 2 or 3 or 4 or more characters". So [23] means "either 2 or 3". What if you wanted to do something similar but with alternative strings of characters, some of which were more than one character long? You would use parentheses again. You would surround all of the choices with a single pair of parentheses, separating the different sequences that match with pipe characters. When a function like re.findall() encounters this in a pattern, it knows to match any pattern that has any one (but only one) of the alternative choices at that point in the pattern.

So, the following pattern captures many different possible titles of professors in a university:

```
r'(Assistant |Associate |Full |Adjunct |Emeritus )?Professor'
```

Question for the reader: Why did we include the trailing space for each item in the group rather than add a single space after it?

Returning to our case citation pattern, we can now capture federal district court opinions too, which are found in the F. Supp. and F. Supp. 2d reporters.

```
r'F\.(Supp\.( 2d)?|([23]d))?'
```

Notice that you can nest parentheses. In this example, the outer parentheses are the ones that control the alternation. While the two pairs of inner parentheses are used for the '?', zero or one, modifiers.

13.5.2 Groups for capturing

Recall the dual purposes of regular expressions: they serve a boolean purpose of identifying a hit and they return a value of the string that triggered the hit. Sometimes, these two purposes are a bit at odds with one another. If you're trying to isolate a particular critical string in a text to return to the user (return value) the only way to do it may be able provide some of the surrounding context before or after the string (boolean purpose). Then, what is returned has extraneous bits at the start and end, which you might wish to trim.

For example, what if you were trying to provide some structure to a form stored digitally, such as the following:

```
Name: Alexander Hamilton DOB: 11/11/1757 POB: Charlestown,
    Nevis
Title: Secretary of the Treasury Political Party: Federalist
Spouse: Elizabeth Schuyler Children: Philip and a bunch more not
    mentioned
```

At this point, you know more than enough to separate out the various fields of information in this file. For example, if you have this data stored as lines in an array, you could identify that you were dealing with the first line with the following regular expression:

```
1 import re
2
```

```
3  text = open('data.txt').read()
4  lines = text.split("\n")
5
6  for line in lines:
7      match = re.search("Name:.+DOB:.+POB:.+", line)
8      if match:
9          print(match.group(0))
10         # Process data further here.
```

In the body of the for loop (at line 9), the command match.group(0) will return the full string that matched. But what you really want is to access the subcomponents of that line. In other words, you want to know that this line has a name "Alexander Hamilton", a DOB "11/11/1757", and a POB "Charleston, Nevis".

The secret is to use parentheses, once again, to separate the pattern into groups. In addition to the modification and alternation functions of parentheses described in the last subsection, parentheses serve the critical role of isolating the different parts of a pattern. Every pair of parentheses is assigned a number starting from 1, reading left to right. So if we rewrote line 7 above to instead say:

```
7      match = re.search("Name:(.+)DOB:(.+)POB:(.+)", line)
```

It would assign whatever matched the '.+' in the first parentheses to the group number 1, the second to group number 2, and the third to group number 3. How do you access these groups? You've already seen the answer, but now you can finally understand it. Remember that the returned match object has a method called .group(). This method takes an integer as its argument, and returns the substring from the original searched text that matched that particular parentheses group. So in the code above:

```
>>> print(match.group(1))
 Alexander Hamilton
>>> print(match.group(2))
 11/11/1757
>>> print(match.group(3))
 Charlestown, Nevis
```

Notice that we've used regular expressions to isolate just the data we wanted. This feature of groups make regular expressions a truly powerful approach to *parsing* data. In one fell swoop, a pattern can identify a very complex pattern in a line of text, and using parentheses and the .group() method, it can isolate just the data part that we care about, leaving behind the surrounding context.

Recall that earlier you used the method match.group(0) with "0" as the argument. When the "0" argument is passed to the .group() method it returns the entire string matched, disregarding the groupings. (It's also important to note that .group() is a method not a list. You use parentheses rather than square brackets to return matching groups.)

You can use the .group() method with any match object, meaning objects returned to .search() or .finditer(). What about .findall()? If you pass a regular expression with grouping parentheses to .findall(), it will return a list of tuples, broken into the groups matched. So if data.txt contained the same form data mentioned above, then the following code:

```
1 import re
2
3 text = open('data.txt').read()
4
5 hits = re.findall("Name:(.+)DOB:(.+)POB:(.+)", text)
6
7 print(hits)
```

would produce:

```
[(' Alexander Hamilton        ', ' 11/11/1757     ', ' Charlestown, Nevis')
 ]
```

To access the various groups of the one (and only) hit, you would use the expressions hits[0][0], hits[0][1], and hits[0][2].

13.5.3 Captured groups and spaces

Once you start using groups to capture data bottled up in unstructured text, you'll have to wrangle spaces. For example, in the previous subsection, we worked with the pattern:

```
7     match = re.search("Name:(.+)DOB:(.+)POB:(.+)", line)
```

producing the following output:

```
>>> print(match.group(1))
 Alexander Hamilton
>>> print(match.group(2))
 11/11/1757
>>> print(match.group(3))
 Charlestown, Nevis
```

Notice that our pattern doesn't contain any spaces, meaning spaces are captured within the string of characters that make up the groups themselves. To illuminate this, we can print characters to show the boundaries of our groups, like so:

```
>>> print("|" + match.group(1) + "|")
| Alexander Hamilton        |
>>> print("|" + match.group(2) + "|")
| 11/11/1757     |
>>> print("|" + match.group(3) + "|")
| Charlestown, Nevis|
```

Odds are that you want to remove the spaces before and after the data (but not the spaces between words in the data). You can do this by post-processing the matched groups, using the `.strip()`, `.lstrip()`, and `.rstrip()` methods introduced in the strings chapter:

```
>>> print("|" + match.group(1).strip() + "|")
|Alexander Hamilton|
>>> print("|" + match.group(2).strip() + "|")
|11/11/1757|
>>> print("|" + match.group(3).strip() + "|")
|Charlestown, Nevis|
```

A second approach is to change the pattern to exclude surrounding spaces from the capturing groups, perhaps using the '\s' character class. For example, this would produce the desired effect:

```
7       match = re.search("Name:\s*(.+?)\s*DOB:\s*(.+?)\s*POB:\s*(.+?)\s*\
            n", text)
```

Notice that this required the use of three non-greedy '+?' modifiers. Try this out on your computer to see what would have happened if those were not marked non-greedy.

13.6 Advanced regex topics

As the final case citation example demonstrates, you can profitably use the regular expression features described so far in this chapter to do all sorts of powerful text processing tasks. You will see, however, that the tools above fall short in some situations. To truly consider yourself a regex expert, you need to understand the features described in this section.

But understand that even though we're calling these topics, "Advanced", they still barely scratch the surface for using Regular Expressions. The web and your favorite bookstore abound with documentation, tutorials, and discussion boards where you can learn even more.

13.6.1 The caret and dollar sign

Two additional, commonly used special characters are the caret '^' and the dollar sign '$'. They refer to the "start of the string" and the "end of the string" respectively. Thus, you'd only ever want the caret at the start of your pattern and the dollar sign at the end of your pattern.

Once again, we have a pattern component that contributes to the boolean purpose of the regular expression–it constrains what counts as a hit–without contributing anything to the return value. A caret or dollar sign doesn't return anything.

```
>>> text = "One ring to rule them all, one ring to find them, one ring the
    bring them all, and in the darkness bind them"
>>> print(re.findall("them", text))
['them', 'them', 'them', 'them']
>>> print(re.findall("them$", text))
['them']
>>> print(re.findall("one", text, flags=re.IGNORECASE))
['One', 'one', 'one']
>>> print(re.findall("^one", text, flags=re.IGNORECASE))
['One']
```

Note that many Python books will introduce you to a companion function to re.search() called re.match(). The only important difference between these functions is that re.match() will return a match object only if the pattern matches at the start of a string. In other words, re.match() implicitly adds a caret to the start of the pattern. Because you can accomplish the same thing using the re.search() function, we prefer just using re.search() with a caret rather than using re.match().

13.6.2 The \b character class

Let's introduce you to one additional character class, '\b'. This example is powerful but potentially confusing. '\b' matches the "boundary" of a word. This means that it will not match any text unless the '\b' falls at the start of a word or the end of a word. In other words, it must fall at the border between whitespace and non-whitespace.

Returning to the example above, notice what this code produces:

```
>>> text = "How much wood would a woodchuck chuck, if a woodchuck,
    would chuck wood?"
>>> print(re.findall(r'wood', text))
['wood', 'wood', 'wood', 'wood']
>>> print(re.findall(r'wood\b', text))
['wood', 'wood']
>>> print(re.findall(r'chuck', text))
['chuck', 'chuck', 'chuck', 'chuck']
>>> print(re.findall(r'\bchuck', text))
['chuck', 'chuck']
```

The first pattern matched "wood" two more times than the second pattern, because the trailing '\b' character class did not match the word "woodchuck". Similarly, the third pattern matched "chuck" two more times than the fourth pattern, because the leading '\b' character class did not match the word "woodchuck".

Once again, just like the caret and the dollar sign, '\b' contributes to the boolean purpose of the regular expression but doesn't contribute anything to the return value.

13.6.3 Regex on strings with more than one lines

In Python, you will often load a multiline text into a single string. For example, the file operations open('filename').read() ingest the entire text of a file, including newlines. You've also been introduced to the '\n' character, which is a textual representation for newlines.

Multi-line strings and the newline characters within them pose two challenges for regular expressions. First, because the '. ' wildcard does not match the '\n' character, patterns will not by default wrap across a newline. Recall the r'vice.president' pattern from earlier in the chapter. This pattern won't match if "Vice" ends a line and "President" starts a new line.

As a quick aside, why is this the default behavior for '. '? One explanation is that regular expressions were devised by people analyzing computer data rather than texts. With many non-textual data file types, each line of the file is a separate record of a single transaction. Web log files, for example, which record information about all visitors to a particular web site, record each user request on a single line. In this context, the default behavior of '. ' is the preferred behavior.

For those using regular expressions to analyze texts, the solution is, once again, the "flags=" keyword argument. The "flag" you need to set is re.DOTALL. So to search for vice.president across newline breaks, you would use:

```
re.findall(r'vice.president', text, flags=re.DOTALL)
```

What if you want this behavior for '. ' but also want to do a case-insensitive search? You can specify more than one flag in the flags keyword argument by separating flags with the "pipe" character, the vertical line produced on most keyboards by pressing shift and the backslash key. You place this between flags, so the full command would be:

```
re.findall(r'vice.president', text, flags=re.DOTALL | re.IGNORECASE)
```

Understand that once you enable re.DOTALL, the '. ' becomes voracious, which might not be what you intend, particularly when paired with repetition. For example, if you added a dot modified by a '+' between "vice" and "president", like so:

```
re.findall(r'vice.+president', text, flags=re.DOTALL | re.IGNORECASE)
```

You would see that this returns only a single hit. That hit's start() position would be the first time "vice" appears in the Constitution, and it's end() position would be after the last time "president" appears, with the '.+' matching the thousands of characters in between.

The second challenge for those who analyze multi-line texts with regular expressions is the behavior of the caret and dollar sign. The default behavior is that the caret matches only the start of the string, thus the very first character, while the dollar sign matches only the end of the string, thus the very last character. Sometimes, it is useful to have the caret and dollar sign match the start or end of a line instead.

Once again, the answer is another flag, re.MULTILINE. The presence of this flag simply changes the behavior of caret and dollar. When present, the caret will match the start of any line, meaning the start of the string or the position immediately following a '\n' character. The dollar sign will match the end of any line, meaning the end of the of the string or the position immediately preceding a '\n' character.

```
>>> poem = """You might belong in Hufflepuff,
... Where they are just and loyal,
... Those patient Hufflepuffs are true
... And unafraid of toil ;"""
>>> print(poem)
You might belong in Hufflepuff,
Where they are just and loyal,
Those patient Hufflepuffs are true
And unafraid of toil ;
>>> print(re.findall("^\S+", poem))
['You']
>>> print(re.findall("\S+$", poem))
['toil ;']
>>> print(re.findall("^\S+", poem, flags=re.MULTILINE))
['You', 'Where', 'Those', 'And']
>>> print(re.findall("\S+$", poem, flags=re.MULTILINE))
['Hufflepuff ,', 'loyal ,', 'true', 'toil ;']
```

Once again, you can activate the behaviors of multiple flags by separating them by pipe characters. It is common in Python programs that process multiline texts to see something like the following:

```
re.findall(r'vice.+president', text, flags=re.DOTALL | re.IGNORECASE |
    re.MULTILINE)
```

13.6.4 re.DOTALL and greedy matching

Once you start using "flags=re.DOTALL", you need to pay special attention to greedy vs. non-greedy matching. Because a '.' will match a newline when this flag is enabled, you might find that a pattern like '.+' or '.*' will ingest many lines or pages of information when you didn't intend it.

For example, let's return to the Constitution. Say you want to take advantage of '.', '+', and "flags=re.DOTALL" to try to isolate a list of the text of all of the Amendments to the Constitution in a compact amount of code. You could try the following (reproducing the three lines of code needed to load the Constitution's text from a text file.)

```
1 import re
2
3 text = open('const.txt').read()
```

```
4
5 matches = re.findall("Amendment \d+.*Amendment \d+", text, flags=re.
      DOTALL)
```

Here, you're trying to take advantage of the fact that every new Amendment starts with the word "Amendment" followed by a one-or-two-digit number and that its text runs until the next time the word "Amendment" followed by a one-or-two-digit number is found.

Unfortunately, if you print the list that results, you'll find a list with only a single element, which starts with the words "Amendment 1" and the text of the First Amendment and runs all the way through the words "Amendment 27" but omits the text of that final Amendment.

There are several things wrong with this code, but let's focus on the problem with the greedy modifier '.*'. Because this is greedy, the star doesn't stop just before the first match of "Amendment \d+", which is the start of the Second Amendment. Instead, it continues to match more and more characters, until it gets to the very last part of the text that matches "Amendment \d+", namely "Amendment 27". Let's add a single fix, changing that modifier to non-greedy, and see what results. This time, we'll print out the first hit as well:

```
1 import re
2
3 text = open('const.txt').read()
4
5 matches = re.findall("Amendment \d+.*?Amendment \d+", text, flags=re.
      DOTALL)
6 print(len(matches))
7 print(matches[0])
```

This results in the output:

```
13
Amendment 1
Congress shall make no law respecting an establishment of
religion , or prohibiting the free exercise thereof; or
abridging the freedom of speech, or of the press; or the right
of the people peaceably to assemble, and to petition the
Government for a redress of grievances.

Amendment 2
```

This is much closer! The list now has 13 hits, almost exactly half the true number. Most importantly, each element now contains just the text of a single Amendment, albeit bracketed by the words "Amendment N" and "Amendment N+1". Not bad for a single character change to the prior code.

How can we perfect this? You don't know quite enough yet. But we'll revisit this again before the end of this chapter.

13.6.5 re.sub()

In addition to re.search(), re.finditer(), and re.findall(), there are two more Python functions you may want to use that operate on regular expressions.

First, re.sub() operates like the string.replace() function we introduced you to earlier. Like .replace(), re.sub() will search-and-replace text that matches. But re.sub() allows you to use a regular expression for the "search" part of this function.

Recall that strings are immutable. re.sub() doesn't change the string you pass it. Instead, it returns a new string, matching the old string but with the specified substitutions made.

Unlike the other re methods we have seen, sub() takes three mandatory arguments: the pattern, the new string you want to use as a replacement, and the string to search.

So consider the following examples:

```
>>> import re
>>> bankRecord="""Client 1: 555-55-5555 Balance $23,400
... Client 2: 555-55-5556 Balance $5,332
... Client 3: 555-55-5557 Balance $0
... Client 4: 555-55-5558 Balance $75,121"""
>>> redactedBankRecord = re.sub("\d{3}-\d{2}-\d{4}", "<redacted>",
    bankRecord)
>>> print(redactedBankRecord)
Client 1: <redacted> Balance $23,400
Client 2: <redacted> Balance $5,332
Client 3: <redacted> Balance $0
Client 4: <redacted> Balance $75,121
```

Notice that re.sub() replaces all matches with the same replacement text.

13.6.6 re.split()

One last Python command for handling regular expressions is re.split(). As the name suggests, re.split() is the regular-expression equivalent to string.split(). It splits the supplied string wherever it finds a match of the supplied pattern. Like the string.split() command, re.split() doesn't include the separator/pattern in the result.

Finally, we're ready to parse the Constitutional Amendments for the last time. What if you wanted to create a list that contained only the text of the Amendments, not the words "Amendment N"? re.split() can do this in a single line. You couldn't do this with ordinary string.split() because the number of the Amendments vary.

But with re.split(), it's as easy as:

```
>>> text = open('const.txt').read()
>>> matches = re.split("\s+Amendment \d+\s+", text)
>>> print(len(matches))
```

```
28
>>> print(matches[1])
Congress shall make no law respecting an establishment of religion , or
prohibiting the free exercise thereof; or abridging the freedom of speech,
    or
of the press; or the right of the people peaceably to assemble, and to
    petition
the Government for a redress of grievances .
>>> print(matches[4])
The right of the people to be secure in their persons, houses, papers, and
 effects , against unreasonable searches and seizures, shall not be violated ,
    and
no Warrants shall issue , but upon probable cause, supported by Oath or
 affirmation , and particularly describing the place to be searched, and the
persons or things to be seized .
```

The zeroth element of the matches list contains all of the text before the First Amendment and can be disregarded or discarded.

Note that both re. split () and re.sub() take all of the same optional keyword "flags" arguments: re.IGNORECASE, re.DOTALL, and re.MULTILINE.

13.6.7 re.VERBOSE

Finally, we've been lamenting this entire chapter how the compactness and complexity of regular expressions make them really difficult to read and understand. Many times in our programming histories, we've revisited our own code just a few weeks or maybe even days after we've written it and have been surprised how baffled we've been deciphering our own regular expressions.

One technique that can help, a little, with this problem is to use the final flag you'll learn for any of the Python functions in this chapter: re.VERBOSE. re.VERBOSE does two things. First, it ignores any whitespace you enter in your pattern, including newlines. This of course means that if you want to match whitespace, you'll need to use the \s character class. Second, any time it encounters the hash (#) mark, it treats the remainder of that line as a comment, and doesn't include it in the pattern.

This allows you to write regular expressions with functionality separated across many lines and liberally annotated with comments. For example, reconsider the case citation regex presented earlier in the chapter, which read:

```
1 matches = findall(r'[1-9]\d{0,2}\sF\.[23]d\s[1-9]\d{0,3}', text)
```

You could use re.VERBOSE to make this a bit easier to read and understand:

```
1 matches = findall(r'[1-9]      # Volume numbers can't start with 0
2                    \d{0,2}    # 1 - 999 covered
3                    \s
4                    F\.        # All begin with ``F.''
5                    [23]d      # 2d and 3d. Doesn't cover bare F.
```

```
6                   \s
7                   [1-9]       # Page number can't start with 0
8                   \d{0,3}     # 1 - 9999 covered
9                   ',
10                text, flags=re.VERBOSE)
```

This doesn't make regular expressions as easy to read as Python code, but it is a big improvement over the uncommented version.

13.7 Conclusion

Regular expressions unlock the power of search. With a few basic rules and some Python library functions, you can build arbitrarily complex and powerful code that can search, manipulate, and repurpose any text you encounter in legal practice.

This marks an important inflection point on your path to becoming a lawyer-coder. Because the practice of law revolves around the handling of texts, you will use regular expressions in a large class of tasks you perform with Python. One way to think of this book is that it is all organized in service of helping you use regular expressions efficiently. You can think of everything in this book up until this point as helping you organize data in order to let you search it with regular expressions. With this view of the world, then, most of what comes later in the book shows you different methods for gathering useful texts–from the web, from services that provide APIs–and the text strings locked up in word documents and excel spreadsheets.

13.8 Cheatsheet

All of the following requires the re library to be loaded first with the command import re.

Using regular expressions in Python

Python provides several functions for searching for regular expression patterns in strings. Three important functions are:

re.search(): find a single match and return a match object The `re.search(pattern, text)` function searches for the first hit that matches regular expression *pattern* in string *text*. The object returned is a *match object*. (See below). For example:

```
1 >>> text = "Particle man, particle man, doing the things a particle can."
2 >>> match = re.search(r'particle', text)
3 >>> print(match.group(0))
4 particle
5 >>> print(match.span())
6 (14, 22)
```

(Remember that these searches are case-insensitive without a flag (see "Flags" below).)

re.finditer(): iteratively find all matches and return match objects The `re.finditer(pattern, text)` function searches for the every hit that matches regular expression *pattern* in string *text*. The hits are *non-overlapping*, meaning that the search for each successive hit begins in the character after the end of the prior hit.

The objects returned are *match objects*. (See below).

`re.finditer()` is typically used in a for loop.

```
1 >>> text = "Particle man, particle man, doing the things a particle can."
2 >>> for match in re.finditer(r'particle', text):
3 ...     print(match.span())
4 ...
5 (14, 22)
6 (47, 55)
```

re.findall(): find all matches and return strings that match The `re.findall(pattern, text)` function searches for the every hit that matches regular expression *pattern* in string *text*. The hits are *non-overlapping*, meaning that the search for each successive hit begins in the character after the end of the prior hit.

This function returns a simple list of strings representing the substrings of *text* that matched. All other information available in a *match object* is not available with this function.

Because this function returns all hits at once, it is not used in a for-loop.

```
>>> text = "Particle man, particle man, doing the things a particle can."
>>> matches = re.findall(r'particle', text)
>>> print(matches)
['particle', 'particle']
```

Match objects

The re.search() and re.finditer() commands return *match objects*. (re.findall() returns a list.)

A match object is a data type that stores the exact substring that matched the pattern but also a lot of useful metadata about the hit. All of this information is accessed using methods. Here are some of the most important methods, but once again, consult the official documentation for a full list.

For all the examples illustrating these methods, assume the match generated with the following code:

```
>>> text = "Don't turn your back. Don't look away. And don't blink."
>>> match = re.search(r'blink', text)
>>> match.start()
49
>>> match.end()
54
>>> match.span()
(49, 54)
```

match.start() This returns an integer, the index of the original string where the match began.

```
>>> match.start()
49
```

match.start() This returns an integer, the index of the original string immediately following the end of the matched characters.

```
>>> match.end()
54
```

match.span() This returns a tuple containing two integers, the indices for the start and one more than the end of the matched characters.

```
>>> match.span()
(49, 54)
```

match.group(0) This returns the entire string that matched the pattern.

```
>>> match.group(0)
'blink'
```

match.group(n) This returns the nth group that matched, reading left-to-right. (See "Groups" below for an explanation and examples.)

Regular Expression syntax

Consult the official Python documentation for the `re` module for the full list of regular expression features Python understands.

Matching Single Characters

Feature	Description	Example
`.`	Matches any character except the newline.	`r'Amendment .'` matches `'Amendment I'` and `'Amendment V'`
`[]`	Matches any one of the characters specified within the square brackets.	`r'Amendment X[IV]'` matches `'Amendment XI'` and `'Amendment XV'`
`[a-z]`	Matches any one of the characters in the range of characters between the character to the left and the character to the right of the hyphen.	`r'subsection [c-f]'` matches `'subsection c'`, `'subsection d'`, `'subsection e'`, and `'subsection f'`
`[^]`	A caret immediately after the opening square bracket matches any one character *except* the characters inside the brackets.	`r'subsection [^c-f]'` matches `'subsection a'`, `'subsection g'`, `'subsection 1'`, and any other character other than those in the range.
`\s`	Matches any whitespace character.	`r'inalienable\s'rights` matches `'inalienable rights'` or `'inalienable\t rights'` but not `'inalienable-rights'`
`\S`	Matches any non-whitespace character.	`r'inalienable\S'rights` matches `'inalienable-rights'` or `'inalienable0rights'` but not `'inalienable rights'`
`\d`	Matches any numeric character.	`r'Item \d'` matches `'Item 1'` or `'Item 9'` but not `'Item A'`
`\D`	Matches any non-numeric character.	`r'Item \D'` matches `'Item A'` or `'Item X'` but not `'Item 3'`
`\w`	Matches any alphanumeric character.	`r'python\w'` matches `'python3'` or `'pythony'` but not `'python!'`
`\W`	Matches any non-alphanumeric character.	`r'python\W'` matches `'python!'` or `'python?'` but not `'python3'`

Repetition

Certain characters in a regular expression indicate repetition. They look for repeated instances of the single character, character class, or group (described below) that immediately precedes them.

Feature	Description	Example
`?`	Repeats the preceding item zero or one times.	`r'Amendment 1\d?'` matches `'Amendment 1'` and `'Amendment 13'` but not `'Amendment 21'` or `'Amendment 111'`
`+`	Repeats the preceding item one or more times.	`r'Amendment \d+'` matches `'Amendment 1'` and `'Amendment 27'` but not `'Amendment One'`
`*`	Repeats the preceding item zero or more times.	`r'Amendment \d*'` matches `'Amendment 1'` and `'Amendment '` but not `'Amendment'` or `'Amendment Two'`
`*`	Repeats the preceding item zero or more times.	`r'Amendment \d*'` matches `'Amendment 1'` and `'Amendment '` but not `'Amendment'` or `'Amendment Two'`
`{m}`	Repeats the preceding item exactly m times.	`r'Amendment \d{2}'` matches `'Amendment 14'` and `'Amendment 20'` but not `'Amendment 1'` or `'Amendment One'`
`{m, n}`	Repeats the preceding item between m and n times (inclusive of both endpoints).	`r'Amendment \d{1, 2}'` matches `'Amendment 1'` and `'Amendment 20'` but not `'Amendment One'`

Greedy and Non-Greedy Repetition

The + and ∗ modifiers will find as many characters as possible that match, stopping only when the part that follows forces it to stop. This is called "greedy matching".

A ? appended onto either of these modifiers instructs it to do "non-greedy matching".

For example:

```
>>> text = "aaaaabbbbbcccccaaaaabbbbb"
>>> match = re.search(r'.+b', text) # greedy
>>> print(match.group(0))
aaaaabbbbbcccccaaaaabbbbb
>>> match = re.search(r'.+?b', text) # non-greedy
>>> print(match.group(0))
aaaaab
```

Escaping special characters

A backslash ('\') in a regular expression will nullify the what the very next character usually does, matching instead just that literal character. So '\.' matches only a period and '*' matches only an asterisk.

Groups

A pair of parentheses in a regular expression surrounds a group, part of a regular expression that can be accessed separately from the entire string that matched using the .group() method of a match object. Groups are numbered by counting open parentheses from left to right.

For example:

```
>>> text = "Dewey, Cheatem, and Howe"
>>> match = re.search(r'(\S+), (\S+), and (\S+)', text)
>>> print(match.group(0))
Dewey, Cheatem, and Howe
>>> print(match.group(1))
Dewey
>>> print(match.group(2))
Cheatem
>>> print(match.group(3))
Howe
```

Groups can also indicate *alteration*, a one-of-many choice of sub-expressions, each of which can contain more than one characters. For example:

```
>>> text = "Department of Commerce"
>>> match = re.search("Department of (Commerce|Defense|State)", text)
>>> print(match.group(0))
Department of Commerce
>>> match = re.search("Department of (Justice|Homeland Security|Energy)"
    , text)
>>> print(match)
```

Finally, groups can also subject more than one character to one of the repetition modifiers above.

```
>>> text = "The last recessions began in 1973, 1980, 1981, 1990, 2001, 2007.
    "
>>> match = re.search(r'(\d{4}, )+', text)
>>> print(match.group(0))
1973, 1980, 1981, 1990, 2001,
```

Flags

Every Python regular expression function takes an optional final keyword argument called 'flags'. The values passed to this argument are constant values from the re library that look like re.DOTALL or re.IGNORECASE.

These flags each change some aspect of what the pattern matches.

To send more than one of the following flags in the same search, combine flags by separating them with the pipe ('|') character. For example, this will send all four of the following flags:

```
1 match = re.search(r'hello', text, flags=re.IGNORECASE | re.VERBOSE |
      re.DOTALL | re.MULTILINE)
```

re.IGNORECASE ignores case

The re.IGNORECASE flags makes the search case-insensitive. This means any alphabet characters in the pattern will match both uppercase and lowercase versions.

re.VERBOSE can produce more readable regular expressions

The re.VERBOSE flag will ignore whitespace, including newlines, and treat the # as a comment to the end of the line. This allows you to comment complex regular expressions.

re.DOTALL for strings with newline characters

Ordinarily, a . in a regular expression will match any character except for a newline. If you pass the constant value re.DOTALL "flag" as the third argument to any Python re function, a dot will match newlines too.

re.MULTILINE for strings with newline characters

Ordinarily, a ^ in a regular expression will match the start of a line and a $ in a regular expression will not work as maybe expected on strings with newline characters. By default, ^ matches only the start of the string and $ matches only the end of the string. If you pass the constant value re.MULTILINE flag as the third argument to any Python re function, a ^ will match the start of any line within a multi-line string (meaning will match at the start of the string or immediately after a newline) and a $ will match at the end of any line within a multi-line string (meaning will match at the end of the string or immediately before a newline).

Other Topics

Matching Special Positions

The following regular expression features constrain where a match can occur but do not themselves return any characters.

Feature	Description	Example
`^`	Matches at the start of a line.	`r'^United States'` finds a match in `'United States v. Jones'` but not `'Katz v. United States'`
`$`	Matches at the end of a line.	`r'United States$'` finds a match in `'Katz v. United States'` but not `'United States v. Jones'`
`\b`	Matches at the start or end of a word (defined as the boundary between an alphanumeric character (e.g. 'a' or '1') and a non-alphanumeric character (e.g. ' ' or '.')).	`r'\bsubstantial\b'` matches `'substantial'` but not `'insubstantial'` or `'substantially'`

re.sub()

The re.sub(pattern, replacement, text) function returns the original *text* but after replacing every substring that matches *pattern* with the string *replacement*.

re.split()

The re.split(pattern, text) function returns a list of the original *text* split at every position where *pattern* produces a hit. The final list does not include the separator/pattern in the result. *replacement*.

Keywords

Case Sensitive—A search that matches both uppercase and lowercase letters.

Character Class—A character followed by a letter that represents a group of characters to match in a regular expression. For example, the character class \s matches whitespace characters.

Character Set—A limited group of possible characters for a single position in a regular expression. Represented by all possible characters surrounded by square brackets.

Escape—Using a \ character before what would otherwise be interpreted as a character with special meaning inside a regular expression (such as \. or \(to match that character alone.

Greedy Matching—The default behavior of the repetition characters $*$ and + to keep matching that keep matching until the part that follows forces it to stop.

Hit—A single match of a pattern in a regular expression.

Match—See hit.

Match Object—The object returned from the re.search() and re.finditer() functions. Contains the text that matched, start and end positions, and other metadata about the match.

Non-greedy Matching—The opposite of *Greedy Matching*. Signified by a ? character immediately after the repetition symbol. Repeats the pattern the minimum times necessary to complete the match.

Pattern—See regular expression.

Regex—See regular expression.

Regular Expression—A search query for strings written according to precise powerful and flexible rules.

Chapter 14

PDF Files

This is a *very* rough draft of this chapter. You won't find nearly as much context as you have in other chapters. You'll also spot a lot of rough prose and possibly even some errors in the examples. Please bear with us! And please let us know if you spot any errors.

We haven't yet written a cheatsheet for this chapter. We hope that the spartan and straightforward nature of this draft makes a cheatsheet a little less necessary.

A true story: One of the authors of this book was chatting with a colleague, a tenured law professor who happened to be married to a senior associate at a "white shoe," "big firm" law firm specializing in corporate law. This law professor was skeptical that lawyers needed to learn to code. "Convince me that you're teaching your law students something that my spouse could use in her legal practice."

The author struggled to impress this skeptic. Bespoke search engine using regular expressions? Yawn. Web scraper to download documents from the Supreme Court's website? Ho hum.

Then, the author described how Python could be used to unlock PDF files. You can use Python and a freely available library to extract text from a PDF file. You can modify a PDF file, pulling pages from one document and inserting them into another. You can transform a virtual pile of dozens or hundreds of PDF documents into a searchable, archivable, manipulable database. In short, you can stop treating PDFs as static images of documents and start treating them as data, subject to analysis and automation.

The skeptic's eyes grew round. He knew that many lawyers regarded PDF as a de facto standard for document interchange, and he knew how working with PDFs could be a cumbersome, slow process. A tool that was useful for unlocking PDF files could create significant efficiencies that even he could not deny.

In this chapter, we will demonstrate how to work with PDF files in Python. You will learn how to extract text from a PDF, reorganize individual pages in a document, and copy pages from document into another.

14.1 Extending Python with libraries

Up until now, we have worked exclusively with built-in Python, with the features and libraries that come with the default installation of the language. Built-in Python does not provide functions for manipulating PDF files, so you must turn to the Internet, to download a new library called PyPDF2. This library is one of thousands that have been created by programmers around the world, who have generously shared the fruits of their labors to extend the language. Throughout this book, we will introduce you to other, similar libraries. The first step is to download and install the library, using the pip command.

14.1.1 The pip3 command

14.1.2 Installing the PyPDF2 library

To begin, install the PyPDF2 library using the pip3 or pip command. Note that capitalization matters for Python libraries.

```
$ pip3 install PyPDF2
Collecting PyPDF2
 Installing collected packages: PyPDF2
Successfully installed PyPDF2-1.26.0
```

If all goes well, you should receive a success message similar to the one above. To test that PyPDF2 is properly installed, start the Python console and enter the following:

```
1 >>> import PyPDF2
2 >>>
```

If Python silently returns you to the console prompt, you know that the installation has succeeded and you are ready to use PyPDF2.

14.2 Basics of PDF Documents

14.3 Working with PDF Documents in Python

In this chapter, we will be working with the published slip opinions of the U.S. Supreme Court. Most U.S. courts release opinions as PDF files on their official websites. Eventually, the text of these opinions end up getting processed by various entities like West and Lexis, who publish hard bound reporters, assign official citations (such as U.S. and L. Ed. for the Supreme Court) and include headnotes. In parallel, the text of these opinions make their way onto various freely available websites run by nonprofits such as Cornell/LII, FastCase, and CourtListener.com. But immediately after an opinion is release, and at least for a little while, the only place to get an opinion is the official website of the court, and those opinions are distributed as PDF files.

At the time of this book's publication, the URL for obtaining Supreme Court slip opinions was https://www.supremecourt.gov/opinions/slipopinion/17, for opinions from the 2017 term. This page contains links to slip opinions dating a bit further back in time, dating back to the 2012 term.

To begin, let's focus on a single opinion, the landmark same-sex marriage opinion, Obergefell v. Hodges, available for download at https://www.supremecourt.gov/opinions/14pdf/14-556_3204.pdf.

Before we turn to Python, open and examine the PDF in your default PDF reader. This is a long, 103 page PDF. The first five pages are the header information and the syllabus. The majority opinion by Justice Kennedy starts on PDF page number 6 (although the human-readable pagination for each part restarts at page 1) and runs to PDF page 39, including two appendices. Each of the four dissenters wrote a separate dissent, Chief Justice Robert's running from PDF pages 40 to 68; Justice Scalia's from 69 to 77; Justice Thomas's from 78 to 95; and finally, Justice Alito's from 96 to 103.

14.3.1 The PdfFileReader object

To begin, download the PDF file from the URL listed above and move it to the current working directory, changing the file's name to obergefell.pdf.

The PyPDF2 library is programmed in a style known as "object oriented". For these purposes, this means that you manipulate a PDF file by associating it with an "object" called a "reader". The object will be referenced with an ordinary Python variable assigned for this purpose. This reader maintains a connection to the content stored in a particular PDF file, and it provides a number of methods you can use to interrogate, examine, and change the data and metadata associated with the file.

To create a PyPDF2 object associated with the Obergefell file you downloaded, do the following:

```
1 import PyPDF2
2
3 reader = PyPDF2.PdfFileReader('obergefell.pdf')
```

This code opens the PDF file and associates it with a reader object accessible through the variable reader. Note that capitalization matters when referring to the PyPDF2 library.

The first thing we can do with this object is determine how many pages the PDF file contains:

```
1 print(reader.getNumPages())
```

When run, this code produces the proper answer for this file: 103.

14.3.2 Working with individual pages

PDF files store contents on individual pages, so the PyPDF2 library also operates a page-at-a-time. The reader object you created in the prior section can't

access individual pages by itself, but instead, it can respond to a method that returns a single page, bundled itself in a Python object called a PageObject. This method is called .getPage() and it takes a single integer, the page you are trying to obtain. The argument is zero-based, so reader.getPage(0) will return the first page, and reader.getPage(reader.getNumPages()− 1) will return the final page. You should probably store the return value in a variable for further processing, like so:

```
1 page = reader.getPage(0)
```

The most useful thing you can do with a PageObject is to extract all of the text into a string with this method:

```
1 page_text = page.extractText()
2 print(page_text[:100])
```

This code produces the following:

```
      1 (Slip Opinion)
OCTOBER TERM, 2014
Syllabus
NOTE
```

These are the first one hundred characters (because of the slice on line 2) of the Obergefell opinion.

At this point, you have converted this PDF file into text, and you can bring to bear all of the string handling and regular expression searching tools you have been learning this semester. Most importantly, PDF files are no longer just images to be viewed with your eyes and printed to a printer. They have become data: searchable, manipulable, and (most importantly) subject to limitless forms of automated processing.

A major caveat about the .extractText() method is needed. Some PDF files are better than others when it comes to extracting text. For example, some PDF files are merely scanned images–essentially photographs–with no text stored in them at all. With this kind of file, the .extractText() method returns nothing. Even PDF files with extractable text will often return text out-of-order, fragmented, or otherwise unusable.

14.3.3 Writing documents with the PdfFileWriter object

The PyPDF2 library can write PDF files as well as read them. The library is probably not up to the task of creating a PDF file out of nothing–you're better off using Microsoft Word or Google Docs for that task. But PyPDF2's write capabilities can help you reconfigure, reformat, or otherwise slice and dice a PDF file into a new format.

Just as you created a "file reader" object to read a PDF file, you create a "file writer" object to write to a PDF. Unlike the reader, however, the writer does not take any arguments during creation. It simply creates an initially empty PDF document. For example:

```
1  writer = PyPDF2.PdfFileWriter()
```

To add content to this open proto-PDF, you can add pages using the writer .addPage() method. But where can you find pages to add? Grab them from another PDF file, using a PyPDF2 reader object! For example, to create a new PDF object that contains just a single page, page 5 of the Obergefell PDF, you would use this code:

```
1  reader = PyPDF2.PdfFileReader('obergefell.pdf')
2  page = reader.getPage(4)   # zero-based, so page 5 of the document.
3
4  writer = PyPDF2.PdfFileWriter()
5  writer.addPage(page)
```

At this stage, the open file writer object is just stored in your computer's RAM memory. How do you write this one-page PDF file to a file on your hard disk? First, you create a new file pointer to an open, writeable file pointer, using the open() function you learned in Chapter 13. Second, use the writer object's .write() method (be careful–not the file pointer's .write() method!) Finally, be sure to close the file pointer. Here is all of this code, in one place:

```
1  reader = PyPDF2.PdfFileReader('obergefell.pdf')
2  page = reader.getPage(4)   # zero-based, so page 5 of the document.
3
4  writer = PyPDF2.PdfFileWriter()
5  writer.addPage(page)
6
7  f = open('page5.pdf', 'wb')
8  writer.write(f)
9  f.close()
```

Two important points about line 7 in this code: First, as always, use special care when writing files with Python. This command will permanently delete the file "pdf5.pdf" in the current working directory without warning! Second, notice that the second argument to the command is 'wb' not just 'w'. This indicates that the file being written is a binary file, as PDF is a binary file format.

14.3.4 Example: Extracting multiple pages from a PDF

Armed with these tools, it's easy to (mostly manually) extract files from PDF file A to write into PDF file B. For example, the following general-use function will take two PDF filenames, from_file and to_file, and start and end pages for the from_file and create a new PDF containing just those pages.

```
1  import PyPDF2
2  import os
3
4  def extract_pages(from_file, to_file, start, end):
5      if os.path.exists(to_file):
6          print("Danger: file {} exists and would be overwritten. Quitting
               function!".format(to_file))
7          return None
8
9      reader = PyPDF2.PdfFileReader(from_file)
10     writer = PyPDF2.PdfFileWriter()
11
12     for page_num in range(start - 1, end):
13         page = reader.getPage(page_num)
14         writer.addPage(page)
15
16     f = open(to_file, 'wb')
17     writer.write(f)
18     f.close()
```

A call to this function of extract_pages('obergefell.pdf', 'syllabus.pdf', 1, 6) will extract just the syllabus and extract_pages('obergefell.pdf', 'majority.pdf', 6, 39) will extract just the majority opinion. (Note that the arguments to the range() function on line 12 were carefully chosen to let the user specify 1-based page numbers, which are probably more natural when talking about PDF files.

14.4 Conclusion

We have just scratched the surface of what you might do with the PyPDF2 library to work with PDF files. Truth be told, the library isn't the most user-friendly one we've encountered, and it lacks detailed documentation. The most important thing you will want to do is use it to extract text from PDF files, as described above, creating strings full of content you can analyze using the other techniques from this book.

Chapter 15

CSV and Spreadsheets

This is a *very* rough draft of this chapter. You won't find nearly as much context as you have in other chapters. You'll also spot a lot of rough prose and possibly even some errors in the examples. Please bear with us! And please let us know if you spot any errors.

We haven't yet written a cheatsheet for this chapter. We hope that the spartan and straightforward nature of this draft makes a cheatsheet a little less necessary.

15.1 CSV files

The closest thing the computer world has to a universal data file format is the CSV file format. CSV files are simply plain text files that obey certain conventions for storing data in rows and columns. They thus bear a close relationship to data stored in spreadsheets, which are also organized by row and column. Probably every major programming language and data-handling program can both read and write data stored in a CSV file.

15.1.1 The CSV file format

CSV stands for "comma separated values". Each line or row in the text file collects the data for a single record, such as the data about a particular person or incident or item. As the (unimaginative) name suggests, the data in a row is further separated by commas. For example, a teacher might store the grades for students in a class in a CSV file that looks like this:

```
Paul,75,72,83
Mary,88,81,90
Jonathan,92,95,100
```

Sometimes, the first row contains the field names. These are the labels that help identify the data in any given row. Reading left to right, the field names

should correspond to the data for that field. So the prior gradebook might have a "header row" that looks like this:

```
Name,Quiz 1,Quiz 2,Final
Paul,75,72,83
Mary,88,81,90
Jonathan,92,95,100
```

Notice two of the field names–"Quiz 1" and "Quiz 2"–contain spaces. Any field on any row may contain a space. After all, it's the commas that signify the end of one field of data and the start of the next. What if, however, you want to enter data that itself might contain a comma? For example, what if the teacher wants to include a comment about each student as a new field on each row? If these comments include (or potentially include) commas, it will be very difficult for a computer program to decipher where one field ends and the next begins:

```
Name,Comments,Quiz 1,Quiz 2,Final
Paul,I talked to him, and he will try harder.,75,72,83
Mary,She was ill, away for interviews, and busy with law review,88,81,90
Jonathan,He's doing really well,92,95,100
```

See how the comments for Paul and Mary pose a problem for a computer program designed to automatically divide these lines into data fields? This is a commonly encountered problem with CSV files, and there is more than one way to address it.

One approach is to choose some other separator–the technical term is *delimiter*–to separate the fields in the rows of the file. (For some reason, we tend to call files CSV files, even if they use a delimiter other than the comma.)

A delimiter can be almost anything, but they tend to be characters that you are not likely to encounter in the data. Some examples include semi-colons ;, pipes |, and tabs \t. A proper CSV file will use one delimiter consistently throughout the file. So the data above with pipes rather than commas will look like:

```
Name|Comments|Quiz 1|Quiz 2|Final
Paul|I talked to him, and he will try harder.|75|72|83
Mary|She was ill, away for interviews, and busy with law review|88|81|90
Jonathan|He's doing really well|92|95|100
```

See how this solves the internal comma problem? It might be, however, that choosing a new delimiter is not enough. You might have data that includes every possible character you might choose as a delimeter. Or you might decide that you want to stick with commas, because they are so widely supported.

In this case, a second solution is to enclose individual data fields in pairs of quote " characters. Once again, these quote characters can be anything, although standard quotation marks are by far most common.

Some CSV files that use quoted fields will put quotations around every single field:

```
"Name","Comments","Quiz 1","Quiz 2","Final"
"Paul","I talked to him, and he will try harder .","75","72","83"
"Mary","She was ill, away for interviews, and busy with law review
    ","88","81","90"
"Jonathan","He's doing really well ","92","95","100"
```

Other CSV files put quotations only around entries that need them:

```
Name,Comments,Quiz 1,Quiz 2,Final
Paul,"I talked to him, and he will try harder .",75,72,83
Mary,"She was ill, away for interviews, and busy with law review",88,81,90
Jonathan,He's doing really well,92,95,100
```

Finally, some CSV files quote certain fields (columns) of the data and not others. They choose which fields to quote by analyzing the data. If even a single data field for a single record of data requires the quotes (due to an embedded comma, for example), then that field will be quoted for every record in the file.

```
Name,"Comments",Quiz 1,Quiz 2,Final
Paul,"I talked to him, and he will try harder .",75,72,83
Mary,"She was ill, away for interviews, and busy with law review",88,81,90
Jonathan,"He's doing really well ",92,95,100
```

15.1.2 Python's csv library: reading CSV files

Python includes a powerful CSV library as part of its "standard library," meaning you don't need to use the pip3 command to start using it. It is called, appropriately, csv, so you need the import csv command in any program that uses it.

The csv library is "object oriented". This means that you read data from or write data to a CSV file by associating the file with an "object" called a "reader". This reader maintains a connection to the content stored in a particular CSV file, and it provides a number of methods you can use to interrogate, examine, and change the data associated with the file.

To create this object association, do the following:

```
1  import csv
2
3  f = open('grades.csv')
4  reader = csv.reader(f)
```

What does this new reader variable contain? How does the csv library convert a two-dimensional CSV file with rows and columns into a Python data type?

The reader variable is a lot like a list-of-lists in Python. The outer list is a list containing other lists. Each inner list corresponds to a single row of CSV data. Within each inner list (row), each element is a single field of data from that row.

This means you can use a for-loop to iterate through the data as follows:

```
1  import csv
2
3  f = open('grades.csv')
4  reader = csv.reader(f)
5
6  for row in reader:
7      print(row)
```

If 'grades.csv' contains the final version of the gradebook in the previous section, this code will produce the following when run:

```
['Name', 'Comments', 'Quiz 1', 'Quiz 2', 'Final']
['Paul', 'I talked to him, and he will try harder.', '75', '72', '83']
['Mary', 'She was ill, away for interviews, and busy with law review', '88',
    '81', '90']
['Jonathan', "He's doing really well", '92', '95', '100']
```

Notice that the code treated the header row as a line of data. We'll show you how to process header row information automatically, in a bit.

When you first call the csv.reader(f) function to create the reader object, you can pass two optional arguments to account for different delimiters and quote characters. These arguments are called, appropriately, delimiter and quotechar. For example, a CSV file that used the tab character as a delimiter and apostrophes as quote characters could be read as follows:

```
1  import csv
2
3  f = open('grades.csv')
4  reader = csv.reader(f, delimiter='\t', quotechar="'")
5
6  for row in reader:
7      print(row)
```

15.1.3 Writing a CSV file

It is also easy to use the csv library to write data to a CSV file. As always, take care when writing data that you not delete files inadvertently!

The csv library has a writer object that is nearly the mirror image of the reader object. To start, open a new, empty file on your disk with the open command, and pass the resulting file pointer to csv.writer(), as follows:

```
1  import csv
2
3  f = open('math.csv', 'w') # Warning: this line will delete this file if it
       already exists!
4  writer = csv.writer(f)
```

Notice that this is slightly different from what you learned about the PyPDF2 library in the last chapter. PyPDF2's object-oriented model has you create an

empty PDF object that is initially not affiliated with any particular file stored on your disk. With PyPDF2, you associate this object with a specific file only after you have filled it with data, and you are ready to write to your computer. In contrast, the csv library's writer object associates itself with a specific open Python file pointer when it is first created.

The writer object has a .writerow() method that will write a row of CSV data at a time. It expects a list as its argument. For example, if you wanted for some reason to store in a csv file the square and cube of every integer from one to twenty, one integer per row, you would use this code:

```
1  i = 0
2  for i in range(1, 21):
3      writer.writerow([i, i*i, i*i*i])
4  f.close()
```

Don't forget to close the file (not the writer) at the end, as in line 3. This will create the following file on your computer:

```
1,1,1
2,4,8
3,9,27
4,16,64
5,25,125
6,36,216
7,49,343
8,64,512
9,81,729
10,100,1000
11,121,1331
12,144,1728
13,169,2197
14,196,2744
15,225,3375
16,256,4096
17,289,4913
18,324,5832
19,361,6859
20,400,8000
```

Like reader, the csv.writer takes the delimiter= and quotechar= arguments. Note that the default behavior of csv.writer is to quote only those lines that need it, meaning none of the lines above would be quoted. To tell csv.writer to quote every field, pass the additional argument quoting=csv.QUOTE_ALL.

15.1.4 Automatically handling header lines with csv.DictReader and csv.DictWriter

The csv library provides another pair of objects called csv.DictReader and csv.DictWriter. They operate nearly identically to csv.reader and csv.writer, respectively, with one key, convenient difference. Rather than treat the data read or written as a list-of-lists, it treats it as a list-of-dictionaries. Even better, for the csv.DictReader at least, it automatically associates the names of the fields for these inner dictionaries using the labels stored in the first, header row of the CSV data.

Returning once again to the 'grades.csv' file that looked like this:

```
Name,"Comments",Quiz 1,Quiz 2,Final
Paul,"I talked to him, and he will try harder.",75,72,83
Mary,"She was ill, away for interviews, and busy with law review",88,81,90
Jonathan,"He's doing really well",92,95,100
```

The program:

```
1  import csv
2
3  f = open('grades.csv')
4  reader = csv.DictReader(f)
5
6  for row in reader:
7      print("{}: {}".format(row['Name'], row['Comments']))
```

would produce the following output:

```
Paul: I talked to him, and he will try harder.
Mary: She was ill, away for interviews, and busy with law review
Jonathan: He's doing really well
```

Notice that inside the for-loop, the data is accessible in a dictionary, allowing you to refer to particular records by their field name, based on their position in the row. This might be far more convenient–and lead to more expressive code–than the bare reader object. Notice further that you didn't need to do anything to tell your code what these dictionary keys should be called. It just inferred it from the header line, which is no longer being treated as data! That's a lot of convenient automation the library provides for very little of your effort!

The csv.DictWriter object is a bit less automatic and flexible, because it needs to know the header field labels in advance, and you need to tell it to write the header line explicitly. So to recreate a subset of the data above you'd use the following code:

```
1  f = open('new_grades.csv', 'w')
2  fields = ['Name', 'Quiz 1']
3  writer = csv.DictWriter(f, fieldnames=fields)
4  writer.writeheader()
5  writer.writerow({'Name': 'Paul', 'Quiz 1':'75'})
```

```
6 writer.writerow({'Quiz 1':'88',  'Name': 'Mary'})
7 writer.writerow({'Name': 'Jonathan', 'Quiz 1':'92'})
8 f.close()
```

This will create the following file on your computer:

```
Name,Quiz 1
Paul,75
Mary,88
Jonathan,92
```

Note that in line 6 of the code, the user supplied a dictionary that flipped the order of the two fields, but DictWriter placed them in the correct order, to make sure they corresponded with the header row.

15.2 Excel files

Spreadsheet programs like Microsoft Excel or Google Sheets have long been the most popular way for storing numeric data such as accounting information or scientific data. In legal practice, you will often receive information like this packaged in a spreadsheet file.

As it happens, a spreadsheet is useful for non-numeric columns. Like CSV files, spreadsheet data is stored in rows and columns. This turns to be a convenient way to organize even textual information. Imagine a spreadsheet tracking electronic discovery in litigation. Each row would refer to an individual document received in discovery, while the columns would track information such as document number, date received, and where it the document is in the review process. Placing this information in a spreadsheet file gives access to powerful functionality such as searching, filtering, and sorting.

Spreadsheet data is also an improvement over CSV because the rows and columns are depicted visually using a graphical user interface, rather than in a jumble of text. (In fact, on most computers, double-clicking a csv file will open it up in a spreadsheet program, breaking the data up into rows and columns automatically.)

Microsoft Excel is probably the most popular spreadsheet program today. It stores its data in files with the extension .xlsx. Native Python cannot read or write xlsx files directly, but there are numerous well-designed libraries that can be used to do so. We'll be exploring a library called openpyxl in this chapter.

15.2.1 Installing openpyxl

To begin, install the openpyxl library using the pip3 or pip command from the shell.

```
$ pip3 install openpyxl
Collecting openpyxl
  Downloading openpyxl-2.5.1.tar.gz (169kB)
```

```
    100% || 174kB 2.5MB/s
Collecting jdcal (from openpyxl)
  Using cached jdcal-1.3.tar.gz
Collecting et_xmlfile (from openpyxl)
  Using cached et_xmlfile-1.0.1.tar.gz
Installing collected packages: jdcal, et-xmlfile, openpyxl
  Running setup.py install for jdcal ... done
  Running setup.py install for et-xmlfile ... done
  Running setup.py install for openpyxl ... done
  Successfully installed et-xmlfile-1.0.1 jdcal-1.3 openpyxl-2.5.1
```

If all goes well, you should receive a success message similar to the one above. To test that openpyxl is properly installed, start the Python console and enter the following:

```
1 >>> import openpyxl
2 >>>
```

If Python silently returns you to the console prompt, you know that the installation has succeeded and you are ready to use openpyxl.

15.2.2 Reading data from an xlsx file

Although xlsx is not as popular as CSV or JSON for exchanging data, there are publicly available datasets distributed as xlsx data online. We'll be using an example from the U.S. Department of Agriculture's Food Safety and Inspection Service. This agency inspects the facilities of meat and poultry producers and sometimes orders recalls. It has posted summary data about these recalls for the years 2008 - 2014 at https://catalog.data.gov/dataset/summary-of-recall-cases-in-calendar-year.

We'll work with the 2014 recall data, so download the file http://www.fsis.usda.gov/wps/wcm/connect/01ae9691-f6d7-41b2-9a19-10cea4927187/FSIS-Recall-Summary-2014.xlsx?MOD=AJPERES

If the downloaded file is double-clicked, it will launch Excel and display the following data:

<Add screenshot here later.>

Note that there are two worksheets in this workbook, labeled (rather unhelpfully) "Sheet1" and "Sheet2". Sheet2 contains a summary of all of the recalls from 2014, while Sheet1 lists details about each individual recall. We'll work with Sheet1, the detailed information.

By convention, we label the columns of a spreadsheet with letters from A to Z (then AA to ZZ, and so on), and we label the rows with numbers starting at 1. A single cell is referred to by concatenating its column letter and row number, so A1, L5, N23, etc. Sheet1 has six columns (labeled A through F) and 96 rows (labeled 1 through 96). The first two rows are header rows, so there are 94 recalls summarized on this sheet.

We will confess our bias at this point: we are big fans of the openpyxl library. Whoever created it is an organizational genius who designed a user interface

that maps Python's language features logically and intuitively to Excel's data storage. For example, consider the following code:

```
1  import openpyxl
2
3  wb = openpyxl.load\_workbook('FSIS-Recall-Summary-2014.xlsx')
4  print(wb.sheetnames)
5  ws = wb['Sheet1']
6  print(ws['A1'].value)
7  %
8
9  If you're an experienced Excel user, you'll understand the approach taken
       above.
10 The topmost level of organization in an excel file is called a \emph{
       workbook}. This is the overarching document that stores all of
11 the information. So too with \lstinline{openpyxl}, which provides a \
       lstinline{load_workbook} function that takes a filename and returns
12 an object referring to the open workbook. Line 3 demonstrates this,
       assigning the open workbook in variable \lstinline{wb}.
13
14 Inside each \emph{workbook} are one or more \emph{worksheets}. This is a
       single ``sheet'' of information, with rows and columns of
15 data stored on it. Again, \lstinline{openpyxl} treats an open \emph{
       workbook} as a Python dictionary, with keys corresponding to the
16 names of stored worksheets that, when referenced, return objects
       representing individual \emph{worksheets}. Again, this is smart,
       inuitive
17 design. So, in Line 4, we use the \lstinline{wb} object to print the value
       of what is known as an \emph{attribute} called
18 \lstinline{wb.sheetnames}. Notice that this isn't a method--there aren't
       any parentheses after \lstinline{.sheetnames}.
19 Instead, it is a variable containing data that the object makes available,
       one that can be accessed using the familiar
20 dot-notation.
21
22 This particular attribute contains a list that contains the names of all of
       the worksheets stored in this workbook.
23
24 In Line 5, we treat \lstinline{wb} like a dictionary, referencing the key `
       Sheet1', which is one of the sheet names in this workbook.
25 This returns a worksheet object, which we store in variable \lstinline{ws}.
26
27 Finally, the \emph{worksheet} object is itself a Python dictionary, which
       can be accessed using the COLROW notation (e.g. `A1') described
28 above. In line 6, we use this to reference the contents of the cell at
       Column A, Row 1, meaning the upper-leftmost cell in the worksheet.
```

29 This returns a cell **or range object**, which has the attribute \lstinline{.value}, which returns the value stored **in** that cell.
30
31 Rather than **access** the data **in** this cell-by-cell fashion, you can also treat the \lstinline{ws} **object** representing a single
32 sheet like a **list** using a **for**-loop **and** an attribute called \lstinline{.values}, as follows:
33
34 %
35 \begin{pythoncode}
36 **for** row **in** ws.values:
37 **print**(row)

The .values attribute lets you access the data in the worksheet associated with ws one row at a time. Each access returns a tuple of information, with the fields of the row appearing in the same order reading left to right.

This code produces the following (abbreviated) output:

```
$ python3 excel.py
['Sheet1', 'Sheet2']
CY 2014 Recalls
('CY 2014 Recalls', None, None, None, None, None)
('Recall Date', 'Recall Number', 'Recall Class', 'Product', 'Reason for
    Recall', 'Pounds Recalled')
(datetime.datetime(2014, 1, 10, 0, 0), '001-2014', 'I', 'Mechanically
    Separated Chicken Products', 'Salmonella', 33840)
(datetime.datetime(2014, 1, 13, 0, 0), '002-2014', 'I', 'Various Beef
    Products', 'Other', 42103)
(datetime.datetime(2014, 1, 15, 0, 0), '003-2014', 'I', 'Beef Franks', '
    Undeclared Allergen', 2664)
(datetime.datetime(2014, 1, 17, 0, 0), '004-2014', 'II', 'Beef and Pork
    Products', 'Undeclared Allergen', 130000)
(datetime.datetime(2014, 1, 17, 0, 0), '005-2014', 'I', 'Spiral Hams', '
    Listeria monocytogenes', 67113)
(datetime.datetime(2014, 1, 19, 0, 0), '006-2014', 'II', 'Cheeseburger Mac
    Products', 'Undeclared Allergen', 1770000)
... many rows deleted...
(datetime.datetime(2014, 12, 29, 0, 0), '092-2014', 'I', 'Beef and Pork
    Products', 'Undeclared Allergen', 38400)
(datetime.datetime(2014, 12, 31, 0, 0), '093-2014', 'I', 'Chicken Products',
    'Undeclared Allergen', 5300)
(datetime.datetime(2014, 12, 31, 0, 0), '094-2014', 'I', 'Canned Soup
    Products', 'Undeclared Allergen', 4474)
```

Note that you need not know the name of the worksheet you want to access, because one worksheet is always designated the "active" worksheet. To work with it, change Line 5 to read:

```
1 ws = wb.active
```

15.2.3 Changing the data in a worksheet

Thanks to the flexible object-oriented nature of this library, it is quite easy to change the data stored in a particular cell. Continuing the code from the previous subsection, this code:

```
1 print(ws['E2'].value)
2 ws['E2'] = 'Disturbing Event'
3 print(ws['E2'].value)
```

changes the value stored in a single cell, E2. Of course, this change isn't saved to the file on disk unless you write the changes, as dicussed in the next section.

15.2.4 Creating a brand new workbook or worksheet

To create a brand-new excel workbook, not associated yet with any file, do the following:

```
1 wb = openpyxl.Workbook()
```

To add a worksheet to this new workbook (or to a workbook you have opened from a file), do the following:

```
1 ws = wb.create\_sheet(sheetname)
2 %
3
4 Where sheetname is the label for the new worksheet.
5
6 \subsection{Writing data to worksheets}
7
8 If you have loaded a file from disk and made changes to it, you can save
      those changes with the following code:
9
10 %
11 \begin{pythoncode}
12 wb.save(filename='test.xlsx')
```

This would overwrite the file 'test.xlsx', if it already exists, so use this command with care.

You can also use this library to create a brand-new Excel workbook, populate it with information (for example, the results of analyses on another data file), and save it. Consider the following code:

```
1 import openpyxl
2
3 wb = openpyxl.Workbook()
```

```
4 ws = wb.active
5 ws. title  = "Westlaw Research Results"
6 ws.append(['Date', 'Case', 'Citation', 'Comments'])
7 ws.append(['6/1/2018', 'Katz v. United States', '389 U.S. 347 (1967)', '
      Created the REP test.'])
8 ws['B3'] = 'United States v. Jones'
9 wb.save(filename='wl-research.xlsx')
```

The ws.append() method used in lines 5 and 6 adds a new row of data below the lowest row containing data in the worksheet. Since this is an empty worksheet, it will add its data to Rows 1 and 2, in this example.

15.3 Conclusion

You will find yourself returning to these two libraries constantly as you continue your development as a Python programmer. CSV is the lingua franca of computerized data exchange, and you'll receive CSV files via email or web download. You will encounter Excel spreadsheets full of information too, especially if you work with financial information.

But just as importantly, you will develop a habit of storing your data into csv or xlsx files. This will help you share your information with co-workers and third-parties.

Finally, because openpyxl is such a well-designed library, you might find yourself gravitating to it as a good catch-all library for data storage, particularly for information that lends itself to two-dimensional organization into columns and rows. Data stored in a spreadsheet file format can be visually inspected with ease, manipulated with powerful spreadsheet tools, and easily turned into graphs and other visualizations.

Chapter 16

Web Scraping

This is a *very* rough draft of this chapter. You won't find nearly as much context as you have in other chapters. You'll also spot a lot of rough prose and possibly even some errors in the examples. Please bear with us! And please let us know if you spot any errors.

We haven't yet written a cheatsheet for this chapter. We hope that the spartan and straightforward nature of this draft makes a cheatsheet a little less necessary.

16.1 How the web works

What happens when you open a web browser, type a URL into the address bar, and hit return?

First, your browser deconstructs the URL into at least three constitutive parts: the part before "://", which is usually https or http; the part after "://" but before the first slash, which is the domain name hosting the web content you are trying to obtain; and the rest, which is the specific, pinpointed web page you are seeking at that domain name.

Your browser sends a web "request" to the domain name using a specialized "protocol" called http, for HyperText Transfer Protocol. We won't say much more about http.

The computer that "speaks web" on behalf of that domain name receives the request from your browser. If it has the specific content you seek, it bundles all of the content up into a "response" and sends it back to your browser. Quite often, that content is organized using what is known as HTML, but more on that in a moment.

If the web sever does not have the content, it returns an error.

16.2 Downloading a web page using Python

16.2.1 Installing the requests library

To begin, install the requests library using the pip3 or pip command.

```
$ pip3 install requests
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Downloading certifi-2018.1.18-py2.py3-none-any.whl (151kB)
    100% || 153kB 2.3MB/s
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Installing collected packages: idna, urllib3, certifi, chardet, requests
Successfully installed certifi-2018.1.18 chardet-3.0.4 idna-2.6 requests
    -2.18.4 urllib3-1.22
```

If all goes well, you should receive a success message similar to the one above. To test that requests is properly installed, start the Python console and enter the following:

```
1 >>> import requests
2 >>>
```

If Python silently returns you to the console prompt, you know that the installation has succeeded and you are ready to use requests.

16.2.2 Obtaining a web page using requests

The requests library may be the best designed and implemented library we will encounter in this book. With very simple, clear, and exception-free syntax, you can use it to obtain the contents of a web page. In other words, it's a stripped-down, Python compatible, non-graphical web browser! For example:

```
1 import requests
2
3 cfaa = requests.get('https://www.law.cornell.edu/uscode/text/18/1030')
4 print(cfaa)
```

Line 3 calls the get() function of the requests library, which takes as an argument the string containing the URL sought.

This code simply prints 200, which is the status code for success. If the URL passed as an argument did not exist, the code would print the familiar 404 code instead.

requests.get() doesn't just print the status code. It returns an object known as a "response object". Similar to the "match object" returned by the regular expressions re library, this object can be used to interact with the content and metadata of the web page returned in many ways.

HTML is simply text, and you can use the .text Python attribute of this object to obtain the HTML from a response object. The following code:

```
1 cfaa = requests.get('https://www.law.cornell.edu/uscode/text/18/1030')
2 print(cfaa.text)
```

Prints the text associated with the webpage to the screen. Here is an excerpt to just the very beginning of what this prints for this page:

```
<!DOCTYPE html>
<html lang="en" dir="ltr"  prefix="og: http://ogp.me/ns# article: http://
    ogp.me/ns/article# book: http://ogp.me/ns/book# profile: http://ogp.
    me/ns/profile# video: http://ogp.me/ns/video# product: http://ogp.me/
    ns/product# content: http://purl.org/rss/1.0/modules/content/ dc: http
    ://purl.org/dc/terms/ foaf: http://xmlns.com/foaf/0.1/ rdfs: http://www
    .w3.org/2000/01/rdf-schema# sioc: http://rdfs.org/sioc/ns# sioct: http
    ://rdfs.org/sioc/types# skos: http://www.w3.org/2004/02/skos/core#
    xsd: http://www.w3.org/2001/XMLSchema#"><!--<![endif]-->
<head profile="http://www.w3.org/1999/xhtml/vocab">
---rest of output omitted---
```

16.2.3 Saving downloaded content to a file

You might want to save content you have obtained using requests.get() to your computer's hard drive. For example, you might want to download all of the PDFs collected on a given web page before you board a long flight, to allow you to read them when you're not online. Or, you might just want to save a copy of the HTML files for a website for long-term storage.

There's a more immediate reason why saving content to local storage is a great idea when writing code that downloads files from the Internet: to ease the burden you are placing on the website's computer server. Now that you're a budding Python programmer, think about how many times you typically run your evolving code when you are developing a program. Once you add a requests.get() line to your code, that code re-downloads content from the same website every time you run it! After the first time you download code, every subsequent download places avoidable load on a web server on the Internet, which at the very least is considered impolite behavior. (In extreme cases, it might even be illegal, a violation of the very law described in the webpage downloaded in the snippet above, 18 U.S.C. 1030, the Computer Fraud and Abuse Act!) It is better to use the techniques described in this part to download the file just once, saving the content to local storage. After that, you can comment out the lines that perform the download, replacing them for the remainder of your testing with simple open() commands. Just don't forget to restore the code that downloads

once you are done debugging your program and ready to share it, deploy it, or turn it in for a grade!

You can use the techniques you've already learned to save the text of a web page to file:

```
1 import requests
2 cfaa = requests.get('https://www.law.cornell.edu/uscode/text/18/1030')
3 f = open('cfaa.html', 'w')     # Deletes contents of file if it already
      exists!
4 f.write(cfaa.text)
5 f.close()
```

You can also use requests to download files stored in non-text binary formats, such as gif, jpg, png, or pdf. To save files like these, you have to use 'wb', which prepares the file for writing in binary, with the open() command. And you need to use the .content attribute of the response object rather than the .text attribute. For example:

```
1 import requests
2 nosal = requests.get('https://cdn.ca9.uscourts.gov/datastore/opinions
      /2016/07/05/14-10037.pdf')
3 f = open('nosal.pdf', 'wb')     # Deletes contents of file if it already
      exists!
4 f.write(nosal.content)
5 f.close()
```

16.2.4 Working with binary file formats using io.BytesIO()

If you are working with binary file formats such as gif, jpg, pdf, docx, or xlsx, you could always save the content directly to a file as described above. If you don't have a need for a long-term copy of the downloaded content, however, you might just want to associate the binary content with a variable, which you can pass to other functions for processing.

For example, if the file you are downloading is a pdf file, you might just want to send it directly to PyPDF2 for further processing and analysis, without ever saving a copy to disk.

In other words, you'd like to do something like the following, but beware: this code won't work as-is:

```
1 import PyPDF2
2
3 data = requests.get(url).content
4 # ERROR! THIS DOESN'T WORK!
5 reader = PyPDF2.PdfFileReader(data)
```

The reason why this fails requires you to delve a little into arcane Python knowledge, but the good news is it's easy to fix.

The problem is that PyPDF2.PdfFileReader() is expecting its argument to be either a filename or an open file pointer. Unfortunately, the result of requests.get() is more akin to the raw binary content of the downloaded file, which is not quite an open file pointer, so PdfFileReader() can't read it.

To bridge this gap, Python provides a bit of glue code called io.BytesIO(). Without going into the full details, this method takes raw binary data and transforms it to look more like an open file pointer.

Say, for example, you wanted to download a PDF into memory (not storage) from a URL using requests.get() and send that PDF data directly into the PyPDF2 library functions. The code to do so would be:

```
1 import PyPDF2
2 import io
3
4 data = requests.get(url).content
5 data_as_file = io.BytesIO(data)
6 reader = PyPDF2.PdfFileReader(data_as_file)
```

Notice that you need to import io (part of the Python standard library so already installed on your computer) to make this work.

16.3 HTML 101

Web pages are written in a language called HTML, for the HyperText Markup Language. Far from a full programming language, a markup language provides a standard set of conventions useful for representing otherwise plain text in a richer manner.

HTML is built around *tags*, small bits of text that begin and end with angle brackets, like so:

```
1 <h1>Hello, world!</h1>
2 Here is my <a href="https://homepage.com"><b>home page</b></a>.<
      br>
3 And here is the <a href="https://school.edu">home page for my school</a
      >.
```

The type or name of a tag is specified by the first character or series of characters after the opening angle bracket. In the example above, there is an "h1" tag in line 1, an "a" tag, "b" tag, and "br" tag in line 2, and another "a" tag in line 3.

Tags often come in matching pairs, with the closing pair indicated by a forward slash before the tag type. The "h1", "b", and two "a" tags above are all paired tags. Matching pairs of tags enclose other tags and non-tag text. On line 2, the opening and closing "a" tag pair encloses both a matching pair of "b" tags and the plain text "home page". We say that the "b" tags are *nested* within its surrounding "a" tags.

Tags serve many purposes, from directing the appearance of part of the website, to creating hyperlinks to other web pages, and more. The html above features four tag types:

h1 - means a "Header, Level 1" tag, which is typically among the largest, most prominent text on the page and tends to be reserved for important titles.

a - a hyperlink. The opening and closing "a" tags surround text that becomes a hyperlink that can be clicked.

b - bold text.

br - a break (like a newline).

Inside an opening tag (meaning after the tag label but before the closing angle bracket) other information may appear, usually in name/value pairs. These are called the "attributes" of the tag. Both of the "a" tags above have href attributes, the first with the value http://homepage.com and the second with the attribute http://school.edu.[1]

Two html tag attributes are especially important for web scraping purposes, the "id" attribute and the "class" attribute. Here is a sample tag that includes both:

For our purposes, the difference between id and class is not that important. As a rough rule-of-thumb, every id should be unique on a given page of html, but you might find the same class more than once, but that's more of a suggestion than a mandatory rule. You're likely to find exceptions.

Believe it or not, that's all the HTML you need to know for now. This barely scratches the surface, and you can find countless online resources that will teach you html. (A popular online resource is https://www.w3schools.com.)

16.3.1 Using your browser's "Inspect" feature to view a web page's HTML

Every modern web browser allows you to "inspect" the HTML of a web page using powerful developer tools. If you right-click anywhere on a web page, and select "Inspect" or "Inspect Element", a panel will appear revealing the HTML behind the page. This is an essential technique for a budding web scraper. You will spend a lot of time poring over the HTML that constitute the websites from which you are trying to extract data.

Some browsers require you to download an inspection tool before you can use this feature. If you don't see the Inspect option when you right-click, Google the name of your browser and "Inspect" to see if special steps are required.

[1] Don't be confused by the terminology here. Python also uses the word "attribute" in a different manner, one we have used in this book from time-to-time. A Python attribute is essentially a variable attached to an object. In the chapter on Excel and the openpyxl library, we noted that the name of the current worksheets was accessible through a Python attribute. We'll try to be clear in this chapter which meaning of attribute we're using.

16.4 Parsing HTML with Beautiful Soup

The task of processing text information to try to divine meaning is known as *parsing*. You could try to parse HTML using regular expressions, but consider for a moment how difficult it would be. For example, think about trying to use regular expressions to parse information from a simple <a> tag such as

Our store

One regular expression that would do no more than pull the "href" attribute from this string is:

r'href=\"(.+)\"'

But this would break if the tag or the enclosed text included a quotation mark, for example:

Our "world famous" store .

Consider yourself lucky that you'll never have to write a regular expression like this. There are many powerful libraries that you can install to search through, manipulate, and otherwise analyze HTML content. We'll be using a fairly powerful library called Beautiful Soup.

16.4.1 Installing Beautiful Soup

To begin, install the Beautiful Soup library, called bs4, using the pip3 or pip command.

```
$ pip3 install bs4
Collecting bs4
Collecting beautifulsoup4 (from bs4)
  Using cached beautifulsoup4-4.6.0-py3-none-any.whl
Installing collected packages: beautifulsoup4, bs4
```

If all goes well, you should receive a success message similar to the one above. To test that Beautiful Soup is properly installed, start the Python console and enter the following:

```
1 >>> import bs4
2 >>>
```

If Python silently returns you to the console prompt, you know that the installation has succeeded and you are ready to use Beautiful Soup.

16.4.2 How Beautiful Soup represents an HTML file

Beautiful Soup is far too complex a library to describe in-depth in this book. The full documentation is available online at https://www.crummy.com/software/BeautifulSoup/bs4/doc/. (Be sure you're reading documentation for version 4 of Beautiful Soup.)

The first step in any program using Beautiful Soup is to create a "soup" object based on some HTML. Here is an example based on HTML downloaded from the web:

```
1 import requests
2 import bs4
3
4 html = requests.get('https://paulohm.com/projects/cplbook/small.html').
      text
5 soup = bs4.BeautifulSoup(html, 'html.parser')
```

This code downloads a very small web page we have put online. It contains little more than the first paragraph of the Wikipedia page for Python, including all of the included hyperlinks.

The soup object that results from this code represents all of the HTML downloaded by the requests library from the URL, but in the form of an object with dozens of powerful methods and Python attributes. For example, you can print the entire page with print(soup).

This outputs:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>A very small page</title>
</head>
<body>
<h1>Python!</h1>
<p>Python is a <a href="https://en.wikipedia.org/wiki/Multi-
    paradigm_programming_language" class="mw-redirect" title="Multi-
    paradigm programming language">multi-paradigm programming
    language</a>. <a href="https://en.wikipedia.org/wiki/Object-
    oriented_programming" title="Object-oriented programming">Object-
    oriented programming</a> and <a href="https://en.wikipedia.org/wiki/
    Structured_programming" title="Structured programming">structured
    programming</a> are fully supported, and many of its features support
    <a href="https://en.wikipedia.org/wiki/Functional_programming" title
    ="Functional programming">functional programming</a> and <a href
    ="https://en.wikipedia.org/wiki/Aspect-oriented_programming" title="
    Aspect-oriented programming">aspect-oriented programming</a> (
    including by <a href="https://en.wikipedia.org/wiki/Metaprogramming"
    title="Metaprogramming">metaprogramming</a><sup id="cite_ref-
    AutoNT-13_40-0" class="reference"><a href="Python_(
    programming_language)#cite_note-AutoNT-13-40">[40]</a></sup
    > and <a href="https://en.wikipedia.org/wiki/Metaobject" title="
    Metaobject">metaobjects</a> (magic methods)).<sup id="cite_ref-
    AutoNT-14_41-0" class="reference"><a href="Python_(
    programming_language)#cite_note-AutoNT-14-41">[41]</a></sup
```

> Many other paradigms are supported via extensions, including <a href
="https://en.wikipedia.org/wiki/Design_by_contract" title="Design by
contract">design by contract<sup id="cite_ref-AutoNT-15_42
-0" class="reference"><a href="Python_(programming_language)#
cite_note-AutoNT-15-42">[42]</sup><sup id="cite_ref-
AutoNT-16_43-0" class="reference"><a href="Python_(
programming_language)#cite_note-AutoNT-16-43">[43]</sup
> and <a href="https://en.wikipedia.org/wiki/Logic_programming" title
="Logic programming">logic programming.<sup id="cite_ref-
AutoNT-17_44-0" class="reference"><a href="Python_(
programming_language)#cite_note-AutoNT-17-44">[44]</sup
></p>
</body>
</html>

You can also strip all of the tags from the html with the method soup.get_text(), which for this web page outputs:

A very small page

Python!
Python is a multi−paradigm programming language. Object−oriented programming and structured programming are fully supported, and many of its features support functional programming and aspect−oriented programming (including by metaprogramming[40] and metaobjects (magic methods)).[41] Many other paradigms are supported via extensions, including design by contract[42][43] and logic programming.[44]

16.4.3 Using Beautiful Soup to search for tags

The soup object isn't just a flat record of the text and html of the webpage. It can be used with powerful search methods that can quickly find and organize tags and tag attributes from throughout the page. For example, often you'll want to isolate every hyperlink in the web page. Remember that hyperlinks are found in <a> tag pairs. To find all of these, use the command:

```
1 print(soup.find_all("a"))
```

This will output the following:

[<a class="mw-redirect" href="https://en.wikipedia.org/wiki/Multi-
paradigm_programming_language" title="Multi-paradigm programming
language">multi-paradigm programming language, <a href="
https://en.wikipedia.org/wiki/Object-oriented_programming" title="

Object−oriented programming">Object−oriented programming, <a
href="https://en.wikipedia.org/wiki/Structured_programming" title="
Structured programming">structured programming, <a href="
https://en.wikipedia.org/wiki/Functional_programming" title="
Functional programming">functional programming, <a href="https
://en.wikipedia.org/wiki/Aspect−oriented_programming" title="Aspect
−oriented programming">aspect−oriented programming, <a href="
https://en.wikipedia.org/wiki/Metaprogramming" title="
Metaprogramming">metaprogramming, <a href="Python_(
programming_language)#cite_note−AutoNT−13−40">[40], <a
href="https://en.wikipedia.org/wiki/Metaobject" title="Metaobject">
metaobjects, <a href="Python_(programming_language)#
cite_note−AutoNT−14−41">[41], <a href="https://en.wikipedia.
org/wiki/Design_by_contract" title="Design by contract">design by
contract, <a href="Python_(programming_language)#cite_note−
AutoNT−15−42">[42], <a href="Python_(programming_language
)#cite_note−AutoNT−16−43">[43], <a href="https://en.
wikipedia.org/wiki/Logic_programming" title="Logic programming">
logic programming, <a href="Python_(programming_language)#
cite_note−AutoNT−17−44">[44]]

Don't be confused. This very messy list appears to just contain snippets of HTML. But notice that there are no quotes around each element–these aren't strings. In reality, each element in this list is itself a full-featured soup object. You can use any of the methods that work on the larger soup object on one of these list elements. For example, this code:

```
1 links = soup.find_all("a")
2 first_link = links[0]
3 print(first_link.get_text())
```

will print

```
multi−paradigm programming language
```

The .get_text() method works, because every element in the list is itself a soup object.

16.4.4 Beautiful Soup and accessing and searching for tag attributes

The find_all() function can also search based on HTML tag attributes. Recall that many HTML tags include "id" and "class" tags. To search for these (or any) attributes, simply use a Python function keyword argument (e.g. id= or href=). But because "class" is a reserved word in Python, you must use the keyword argument "class_" instead.

Continuing the code from above, the following snippet:

```
1 print("ID tags:")
```

```
2 id_tags = soup.find_all(id="cite_ref-AutoNT-14_41-0")
3 print(id_tags)
4 print()
5
6 print("Class tags:")
7 class_tags = soup.find_all(class_="reference")
8 print(class_tags)
```

will print

```
ID tags:
[<sup class="reference" id="cite_ref-AutoNT-14_41-0"><a href="
    Python_(programming_language)#cite_note-AutoNT-14-41">[41]</
    a></sup>]

Class tags:
[<sup class="reference" id="cite_ref-AutoNT-13_40-0"><a href="
    Python_(programming_language)#cite_note-AutoNT-13-40">[40]</
    a></sup>, <sup class="reference" id="cite_ref-AutoNT-14_41-0"><
    a href="Python_(programming_language)#cite_note-AutoNT
    -14-41">[41]</a></sup>, <sup class="reference" id="cite_ref-
    AutoNT-15_42-0"><a href="Python_(programming_language)#
    cite_note-AutoNT-15-42">[42]</a></sup>, <sup class="reference"
    id="cite_ref-AutoNT-16_43-0"><a href="Python_(
    programming_language)#cite_note-AutoNT-16-43">[43]</a></sup
    >, <sup class="reference" id="cite_ref-AutoNT-17_44-0"><a href="
    Python_(programming_language)#cite_note-AutoNT-17-44">[44]</
    a></sup>]
```

.find_all() will also accept more than one argument, so you can search for a tag type, id, and class all at once by passing all three as arguments. Doing so will match only tags that meet all three criteria.

Once you've isolated a tag through a search, you can also access the tag's attributes. Each soup object returned by find_all can be treated just like a dictionary in which every attribute/value pair in the tag is turned into a Python key/value pair. For example, the following code:

```
1 links = soup.find_all("a")
2 first_link = links[0]
3 print(first_link['href'])
```

will print

```
https://en.wikipedia.org/wiki/Multi-paradigm_programming_language
```

You can also produce the entire dictionary of HTML attribute/value pairs by referring to soup.attrs:[2]

[2]This is the high water mark of confusion about the double-meaning of the word "attribute". soup.attrs is Python code that refers to the Python "attribute" of the soup object. This

```
1 links = soup.find_all("a")
2 first_link = links[0]
3 print(first_link.attrs)
```

will print

```
{'class': ['mw-redirect'], 'href': 'https://en.wikipedia.org/wiki/Multi-
    paradigm_programming_language', 'title': 'Multi-paradigm
    programming language'}
```

16.4.5 Relative and absolute URLs

Recall that when you learned about file paths for files stored on your computer, you learned that file paths can be absolute–meaning they can specify a full, unique path from the "top" of your hard drive's folder structure to the file in question (e.g. /Users/ohm/Documents/python/ps6/brandeis.txt)–or they can be relative–meaning they specify a partial path relative to the current working directory (e.g. brandeis.txt or ../brandeis.txt).

URLs can also be absolute or relative. An absolute URL is a URL that includes the full location of a file on the web, including the domain name of the server hosting the data. So http://paulohm.com/classes/cj16/index.html is an absolute URL.

Relative URLs aren't relative to a "current working directory" but instead are relative to a "base URL", meaning the URL for the web page that led you to the relative URL, usually through an <a> hyperlink.

For example, let's say you used requests.get() to download the content at http://paulohm.com/classes/cj16/index.html. In the HTML that you downloaded, you would find this html link:

<a href="files/ExcerptFloridaJardines.pdf">

Notice that the URL in this link doesn't begin with http: and doesn't include a domain name. This is a relative URL. If you sent this relative URL directly to requests.get(), it would result in an error, because requests.get() expects an absolute URL.

Turning a relative URL into an absolute URL isn't a straightforward process, but luckily there's a Python library that will do it for you that doesn't require any thought on that on your part. We'll introduce that library in the next subsection, but for completeness, here are the rules for turning a relative URL into an absolute URL.

Rule 1: If the relative URL begins with a slash (e.g. <a href="/courses.shtml"), this means "append this to the domain name part of the base URL." So if the base URL, once again, is http://paulohm.com/classes/cj16/index.html, then this relative URL translates to http://paulohm.com/courses.shtml.

Rule 2: If the relative URL has no leading slash, append it to the final slash in the base URL, omitting anything in the base URL that comes af-

dictionary itself contains the HTML attributes (and values) for the HTML represented by these objects. Confused?

ter the final slash. So if the base URL is http://paulohm.com/classes/cj16/index.html and the relative URL is <a href="files/ExcerptFloridaJardines.pdf">, the resulting absolute URL is http://paulohm.com/classes/cj16/files/ExcerptFloridaJardines.pdf.

There are other wrinkles, but they go beyond the scope of this book.

16.4.6 Turning all URLs into absolute URLs with urllib.parse.urljoin()

The distinction between absolute and relative URLs matters, because you should only send URLs to methods like request.get() to download content from the web. Sending a relative URL will almost always result in an error. Happily, Python makes it easy to turn any relative URLs into their absolute equivalents, using a function called urllib.parse.urljoin(). You must start by importing the urllib.parse library, which comes standard with Python. Then, any time you download content from url1 containing a link to url2, which might be relative, you can ensure that url2 is absolute using the single command:

```
1 import urllib.parse
2
3 url2 = urllib.parse.urljoin(url1, url2)}
```

This function works regardless if url2 is absolute, relative without a leading slash, or relative with a leading slash. The library knows what to do in every case, saving you the trouble of dealing with the subtleties yourself.

16.5 From Scraping to Crawling the Web

Until now, all of the web content we have downloaded and searched through has originated from a URL you have supplied manually to the code. You have usualled grabbed these URLs by copying them from the URL bar of your browser. But much of the interesting data on the web is *dynamic*, meaning it is not tied to one, simple, static URL. You need to learn additional features of the requests library in order to learn how to access dynamic content. And you need to study strategies for extracting dynamic content from these pages using Beautiful Soup and your browser's "Inspect" capability.

16.5.1 Web forms and URL query strings

Dynamic content is often tied to what are known as *web forms*. You know what these are–you use dozens of them every day. A web form is the generic name for any portion of any web page that permits user input or interaction, from input fields to radio buttons, drop-down menus, checkboxes, and submit buttons. The familiar Google search engine's home page is perhaps the web's most famous web form. Visit it now.

<Insert image of Google home page>

If you right click on this page and select "Inspect," you will see the HTML for this page. From the Inspect panel, click Command-F (on a Mac) or Control-F (in Windows) and search for the characters <form. You should find only one hit on the page, which begins:

<form class="tsf" action="/search" style="overflow: visible " id="tsf" method ="GET" name="f" onsubmit="return q.value!='' " role="search">

This HTML tag delineates the start of Google's web form, which consists of *a lot* of other HTML up-to-and-including the closing </form> tag. In the opening tag, pay special attention to the method="GET" attribute. There are two types of forms on the web, called GET and POST. You interact with the two types differently, so keep this in mind. More on this much later.

If you hunt through this mess of HTML, you'll find two particularly interesting tags, both of type <input>:

<input class="gsfi" id="lst-ib" maxlength="2048" name="q" autocomplete ="off" title="Search" type="text" value="" aria-label="Search" aria-haspopup="false" role="combobox" aria-autocomplete="list" style=" border: none; padding: 0px; margin: 0px; height: auto; width: 100%; background: url(" AMDAwAAAACH5BAEAAAAALAAAAAABAAEAAAICRAEAOw%3D %3D") transparent; position: absolute; z-index: 6; left: 0px; outline : none;" dir="ltr" spellcheck="false">

<input value="Google Search" aria-label="Google Search" name="btnK" type="submit" jsaction="sf.chk">

The first tells your browser to draw the text field where you enter your search query. Looking through the attributes of this tag, you can see that it is type="text", which marks this as a text input field, and name="q", which we'll see again in a second.

The second is the prominent "Google Search" button itself. Again, the attributes reveal both that this is a button as well as the label that appears on the button.

When you enter a search term (say "Python") and click "Google Search", your browser loads the familiar search query results, pictured here:

<insert query results for "Python" search>

Take a close look at the URL that now appears in the URL bar of your browser:

https://www.google.com/search?hl=en&source=hp&ei= LrLCWsDtGua2ggeBnrzAAQ&q=Python&oq=Python&gs_l=psy-ab .3..0j0i131k1j0l4j0i131k1j0l3 .464979.465486.0.465567.11.7.0.0.0.0.78.370.5.6.0....0...1.1.64.psy-ab ..5.5.369.0...79.NfbfA4dFayk

You'll see something slightly different in your browser, because there is a lot of variability in how Google works from computer-to-computer and day-to-day. But let's break this URL into its constituent parts.

First, comes https://. This indicates that this is a web page that uses privacy-protective encryption for communication (indictated by the https rather than http).

Second, comes the domain name: www.google.com/.

Third, is the relative path: search.

But see the question mark after that? That specifies that the rest of the URL is what we call a "query string". It looks like gobbledeegook, but look what happens if you treat the ampersands that are sprinkled throughout as separator marks. This is what results:

```
hl=en
source=hp
ei=LrLCWsDtGua2ggeBnrzAAQ
q=Python
oq=Python
gs_l=psy-ab.3..0j0i131k1j0l4j0i131k1j0l3
        .464979.465486.0.465567.11.7.0.0.0.0.78.370.5.6.0....0...1.1.64.   psy-ab
      ..5.5.369.0...79. NfbfA4dFayk
```

Does this remind you of anything in Python? They look a bit like variable assignments or dictionary entries, connecting a label on the left (say, source or q) with data on the right (hp and Python). In the context of a query string, we'll call the items on the left "parameters" rather than keys.

The parameters in the query string are like function arguments or user input to the Google search engine. We can make some educated guesses about some of them. For example hl=en might mean "English". To test this, try to replace it with hl=fr. (If you don't have this parameter in your URL, just add it, but be sure it is separated by its neighbor parameters by ampersands.)

If all goes well, you'll see French all over the page. Try replacing "fr" with "es" or "ru". What results?

But of much more interest is q=Python and oq=Python. These seem to match the search query you entered, meaning it is the parameter used to tell Google what you're searching for. Try replacing "Python" with another one-word search query, and hit enter. Sure enough, you can now change the search query directly from the URL bar. If you experiment more, you'll see that you can enter multi-word phrases as search queries by using the + character instead of spaces, so q=Python+requests+library.

Think back to the HTML behind Google's web form. Remember the <input > tag you looked had an attribute name="q"? It's no accident this same label showed up both in the web form's HTML and in the query string.

This is the essence of web scraping a web form: First, you hone in on the relevant pair of <form></form> tags. On a complicated web page with lots of user interaction, there might be many such forms. By inspecting the HTML behind the web form, you identify all of the name= labels for any relevant input fields you need to "fill in". You use those labels to construct the proper parameter/value pairs in the query string that results.

But this is all prelude to the code. How do you use Python to scrape a web page that is dynamically generated with a web form?

16.5.2 Submitting web form data in Python

Let's practice submitting data using a web form with a different example, one a bit more closely related to legal practice. The Occupational Safety and Health Administration (OSHA) performs enforcement inspections of workplaces for health and safety violations. They allow the public to search their database of inspections they have performed at the URL https://www.osha.gov/pls/imis/establishment.html. The web page consists of a simple form that looks like the following:

<screenshot here>

A good first step in web scraping the data behind a web form is to just try a manual search in your browser. Enter the word "pizza" in the "Establishment" field, don't change anything else, and click "Submit". You'll see results that look something like this:

<screenshot>

More importantly, you'll see that the URL has changed to something like:

https://www.osha.gov/pls/imis/establishment.search?p_logger=1&
establishment=pizza&State=all&officetype=all&Office=all&p_case=all&
p_violations_exist=all&startmonth=04&startday=02&startyear=2013&
endmonth=04&endday=02&endyear=2018

This should seem familiar to you. This question mark in the URL delineates the start of the query string, which is made up of numerous parameter/value pairs, separated from its neighbors by ampersands. The tell-tale parameter seems to be establishment=pizza.

Notice that the other pairs in this URL are a lot easier to decipher than Google's mysterious parameters. Whenever you are trying to reverse engineer a URL in this manner, and you have isolated what you think is the key parameter (or two or three), you are left with a choice of what to do with the rest of the parameters, which might be more cryptic and maybe een totally incomprehensible. Your choices are:

1. Omit them.
2. Replicate them, as-is.

The problem with strategy 1 is that some of the other parameters might be necessary for the search to work at all. The problem with strategy 2 is that some of the parameters might be uniquely tied to the search you've just done, and they won't work for another search.

The conservative path is to start with strategy 2 and include every other parameter, verbatim, except the one or two you think you need to change. If that doesn't work, switch to strategy 1. And remember that we're not even talking about Python yet: you can edit the URL and reload the page directly in your browser URL bar itself, before you write a line of code. Some people call this act of manipulating a URL's query string, "URL munging".

If you wanted, you could try to hard-code this entire URL, question mark and ampersands and all, into a string that you would pass directly as the first argument passed to the requests.get(). You could even try to build up the URL using string concatenation or the string .format() method. So to build up the URL above, searching for "pasta" rather than "pizza", you could do something like this:

```
1  query = "pasta"
2  url = "https://www.osha.gov/pls/imis/establishment.search?p_logger=1&
       establishment={}&State=all&officetype=all&Office=all&p_case=all&
       p_violations_exist=all&startmonth=04&startday=02&startyear=2013&
       endmonth=04&endday=02&endyear=2018".format(query)
```

It is perfectly acceptable to munge URLs in this way. But we think you'll more often prefer to use another, simpler, more fool-proof method, using another feature of requests.get().

requests.get() accepts an optional, second argument, in addition to the URL, called params=. This argument must point to a dictionary full of keys and values that correspond to the parameter/value pairs in the query string. You'll typically pre-populate that dictionary before you send it as the params argument.

Behind the scenes, requests.get() will take the params dictionary, construct the query string, and "attach" it to the URL it uses to fetch the content.

Here is some code that will use this method to download results of a search to the OSHA website:

```
1  import requests
2  import bs4
3  import urllib.parse
4
5  query=input("Search term? ")
6  query = query.replace(" ", "+")
7  url_base = "https://www.osha.gov/pls/imis/establishment.search"
8  parameters = {'establishment': query}
9  response = requests.get(url_base, params=parameters)
10
11 print(response.text)
```

Notice a few things about this code. First, we chose "strategy 1" in line 8. The parameters dictionary we generated contained just the one parameter we knew we needed to conduct the search.

Second, through trial-and-error, we determined that, like Google, this website requires + marks in place of spaces in the search query string, explaining line 6.

Finally, line 11 prints the HTML that the website outputs. Scrolling back through it, we can see that this worked!

16.5.3 Using Beautiful Soup to extract records from the OSHA web page

Now let's finally put to work some of the Beautiful Soup methods you learned earlier in this chapter. Once again, use "Inspect" or "Inspect Elements" in your browser to look at the web page of search query results you received:

Here is an excerpt of the first few dozen lines of this data:

```
<table class="table table-bordered table-striped">
<thead><tr>
<th>&amp;nbsp;</th>
<th>#</th>
<th>Activity</th>
<th>Opened</th>
<th>RID</th>
<th>St</th>
<th>Type</th>
<th>Sc</th>
<th>SIC</th>
<th>NAICS</th>
<th>Vio</th>
<th>Establishment Name</th>
</tr></thead>
<tbody><tr>
<td><input type="checkbox" name="id" value="1292609.015"></td>
<td>1</td>
<td><a href="establishment.inspection_detail?id=1292609.015" title
    ="1292609.015">
<em>1292609.015</em></a></td><td>02/05/2018</td>
<td>0950615</td>
<td>CA</td>
<td>Planned</td>
<td>Complete</td>
<td></td><td>311824</td><td>&amp;nbsp;</td><td>Pasta Sonoma, Llc
    </td>
</tr>
<tr>
<td><input type="checkbox" name="id" value="1270393.015"></td>
<td>2</td>
<td><a href="establishment.inspection_detail?id=1270393.015" title
    ="1270393.015">
<em>1270393.015</em></a></td><td>10/05/2017</td>
<td>0213900</td>
<td>NJ</td>
<td>Complaint</td>
<td>Complete</td>
```

```
<td></td><td>311412</td><td>5</td><td>Caesars Pasta Llc.</td>
</tr>
```

This is an unusual bit of HTML. Notice that none of the tags include any attributes, except for the type="checkbox" attributes in <input> tags and the href= attributes in <a> tags. This means that you won't be able to hone in on a particular "id" or "class" that isolates your data.

Notice also that it helps to know a little HTML when scraping, although you might be able to intuit what is going on here. In HTML, a table of information is bracketed by <table> tags. Each row is surrounded by <tr> tags, and each column within each row is surrounded by <td> tags.

Since you want to isolate all data relating to a single workplace, you want to isolate a single row, bracketed by <tr> tags, such as the following:

```
<tr>
<td><input type="checkbox" name="id" value="1292609.015"></td>
<td>1</td>
<td><a href="establishment.inspection_detail?id=1292609.015" title
    ="1292609.015">
<em>1292609.015</em></a></td><td>02/05/2018</td>
<td>0950615</td>
<td>CA</td>
<td>Planned</td>
<td>Complete</td>
<td></td><td>311824</td><td> </td><td>Pasta Sonoma, Llc
    </td>
</tr>
```

Notice that there are twelve <td> pairs in this snippet. This corresponds to the twelve columns in the table on the page of results.

You know how to isolate this, by turning the HTML into a soup object and then using the soup.find_all() method. Continuing from the code above:

```
1 response = requests.get(url_base, params=parameters)
2
3 soup = bs4.BeautifulSoup(response.text, 'html.parser')
4
5 rows = soup.find_all("tr")
6 print(len(rows))
```

Running this code reveals that it found 24 <tr> tag pairs in the code. This wasn't expected, because the results in your browser stop at 20 hits. What happened? Examining the HTML again reveals the source of the false positives. Look, for example, at these lines of HTML from much earlier in the page source:

```
<h2>Establishment Search Results</h2>
<div class="table-responsive"><table class="table table-bordered table-
    striped">
```

```
<tr><th>Establishment</th><th>Date Range</th><th>Office</th><th
    >State</th></tr>
<tr><td>pasta</td><td>04/02/2013 to 04/02/2018</td><td>all</td
    ><td>all</td></tr>
</table></div>
```

This small table appears above the table with the results and explains two of the False Positive <tr> tags. To isolate just the True positives, we take advantage of our knowledge that the rows we want have twelve columns, while this table has far fewer. We can thus test each row and report only the ones with many columns. We'll see the code to do this below in a moment.

But what should we do with these isolated workplaces? For now, let's use an additional .find_all() method (remember that each match from the first search is itself a Beautiful Soup object that can be searched through further) to look for two key pieces of information: the name of the restaurant (stored in the 11-th <td> pair, and the date of the inspection, in the 3-th td pair. Putting this all together results in the following code, picking up once again from where the last code left off:

```
1 rows = soup.find_all("tr")
2 #print(len(rows))
3
4 for row in rows:
5     cols = row.find_all('td')
6
7     if len(cols) > 10:
8         print("{}\t{}".format(cols[3].get_text(), cols[11].get_text()))
```

As of the day this chapter was written, this returns the following results:

```
$ python3 osha.py
Search term? pizza
03/22/2018      Vino'S Pizza Grill
02/22/2018      Authentic Pizza Llc
02/13/2018      Nation Pizza Products, L.P.
02/02/2018      Posti'S Pizza
01/19/2018      Princess & Sons Pizza, Inc.
01/09/2018      Ozark Pizza Company, Llc
01/05/2018      Ko Olina Pizza Corner Llc
01/03/2018      Lnzro Pizza Empire, Inc. Dba Lorenzo Wholesale Foods
12/27/2017      Team Arizona Pizza, Inc.
12/21/2017      Team Arizona Pizza, Inc.
12/20/2017      Villarreal Pizza, Inc.
12/13/2017      Lewisburg Pizza, Llc
11/29/2017      Di Lauro'S Bakery & Pizza, Inc
11/28/2017      Pizza Hut Of Southeast Kansas, Inc
11/22/2017      Domino'S Pizza
11/21/2017      Wa317947343 - Mod Super Fast Pizza Llc
```

11/14/2017	Mountain View Pizza Company
11/07/2017	Sals Pizza Brentwood Inc
10/04/2017	96162 − Parrish Enterprises Inc Dba Papa Murphys Pizza
09/26/2017	California Pizza Kitchen, Inc.

16.5.4 Crawling through the results

We graduate from mere scraping to *crawling* when we write code that can access more than one page of data, by finding the hyperlinks inside <a> tags that will advance to the second (and subsequent) page, maybe even moving from one website to another. Some crawlers work perpetually, following links forever, in theory eventually visiting every reachable website on the web. We tend to call these programs *spiders*.

For example, look again at the data we have scraped from the OSHA search query result page:

<screenshot again>

Notice that each row in the table (in the 2-th element of the cols list we generated above) contains a hyperlink. Clicking on the first of these pulls up a differen website that looks like this one:

<new screenshot>

This page lists a bit more information about each inspected workplace. For example, only by clicking on this link could we learn the full mailing address for the workplace.

Notice also that the URL for this detail page is again a URL with a query string: https://www.osha.gov/pls/imis/establishment.inspection_detail?id=1292609.015 . But these URLs aren't dynamically generated URLs based on the contents of a web form. Instead, these URLs are just sitting inside the href attributes of <a> tags on the prior page. (See the HTML snippet listing for "Pasta Sonoma, Llc", above.)

Armed with this knowledge, let's try to beef up the output from the example above. Now, for each inspected workplace, let's find the URL that links to the specific related detail page; visit that page; find the mailing address for the workplace; and report that (along with the date of inspection and name).

This is an example where it might be simpler to use a regular expression than Beautiful Soup (although this might be a question of personal preference), because the detail page is relatively well-structured, and appears to include the words "Mailing:" on the same line as the entire mailing address. It's a good reminder that you're always free to mix regular expressions and Beautiful Soup if you please. The new code looks like the following (remembering that this is a continuation of earlier code, and you'd need to add a new import re to make this work).:

```
1 rows = soup.find_all("tr")
2 print(len(rows))
3
4 for row in rows:
```

```
5       cols = row.find_all('td')
6
7       if len(cols) > 10:
8           a_tags = cols[2].find_all('a')
9           href = a_tags[0]["href"]
10          response2 = requests.get( urllib.parse.urljoin(url_base, href))
11          match = re.search(r'Mailing: (.+?)</td', response2.text)
12          if match:
13              mailing = match.group(1)
14          else:
15              mailing = "unknown"
16
17          print("{}\t{}\t{}".format(cols[3].get_text(), cols[11].get_text(),
                mailing))
```

Line 10 downloads the detail page (remembering to use urllib.parse.urljoin() to turn the relative URL into an absolute URL); Line 11 is the regular expression; and Lines 12-15 deal with potential False negatives. Line 17 prints the results.

16.5.5 More crawling: grabbing more pages

Most website search queries return data a page at a time, to make the results load more quickly and to make the data more manageable for the end user. With results like these, a very common task is automatically loading each subsequent page of data until all of the data has been loaded.

We approach this task essentially the same we approached clicking on the detail pages above. In the browser, we find the "next page" link or button, which in this case is a small section that says Result Page: 1 2 3 > where each number and the > symbol load various pages. Clicking the > symbol loads the next page of results, at this URL:

https://www.osha.gov/pls/imis/establishment.search?establishment=pasta&state=all&officetype=all&office=all&startmonth=04&startday=02&startyear=2013&endmonth=04&endday=02&endyear=2018&p_case=all&p_start=0&p_finish=20&p_sort=12&p_desc=DESC&p_direction=Next&p_show=20&p_violations_exist=both

It takes a fair amount of studying (especially comparing this to the URL generated for the first page of results), but after a lot of trial-and-error, you'll isolate these three parameters:

p_start=0
p_finish=20
p_direction=Next

To load the second page of data, then, requires a slightly larger parameters dictionary than we used previously: parameters = {'establishment':query, 'pstart': 0, 'pfinish': 20, 'pdirection':'Next'}. To load the third page, you'd

pass the dictionary parameters = {'establishment':query, 'pstart': 20, 'pfinish': 40, 'pdirection':'Next'}. Etc, etc.

We'll leave the rest as an exercise for the reader.

16.5.6 POST queries

Let's quickly look at another page you might want to search. Take a look at the web form on the website of the Federal Election Commission (FEC) used to search for campaign finance contributions to political action committees at https://classic.fec.gov/finance/disclosure/norindsea.shtml. The page looks like the following:

<screenshot>

Once again, use the web form from your browser to see what results. This is a very simple form, containing just fields for Last Name and First Name, a "Get Listing" button and a "Clear Form" button.

Entering a valid first and last name returns a page of results, but the URL looks like this: http://docquery.fec.gov/cgi-bin/qind/

What happened? Return to the web form page and use "Inspect" to look at the HTML behind the web form.

Here is the complete HTML for the very short form:

```
<form method="POST" action="http://docquery.fec.gov/cgi-bin/qind/">
<!--FORM METHOD="POST" ACTION="http://docdocquery.fec.gov/cgi
    -bin/qind/"-->
  <p> Individual Name (<i>Last, First</i>):  
     <input type="text" name="lname" size="20" maxsize="100">
     <input type="text" name="fname" size="20" maxsize="100">
      <br>
      <br>
      <input name="submit" type="submit" value="Get Listing">
     
    <input name="reset" type="reset" value="Clear Form">
    <br>
  </p>
  <p>
  No Luck?? Try an <a href="advindsea.shtml">Advanced Search</a><br>
  </p>
 </form>
```

Notice the <form> tag has one key difference from the Google and OSHA search forms. Those forms both had an attribute of method="GET", while this form says METHOD="POST". (both the attribute name and the values are case insensitive).

The key difference between GET and POST is the one you've already noticed. While GET queries cram all of the form information into a long query string that is packed onto the end of the URL, POST queries instead send the

parameters directly to the web server without using the URL. There are other differences that are not nearly as important to would-be scrapers and crawlers.

The great news is that there is only one important difference from Python: Use the requests.post() method rather than requests.get(). Both take the URL as the first argument and the params= argument as the second argument.

But if you can't reverse engineer a URL, how do you know what parameters to pack inside the params= dictionary? As you've probably guessed, you study the HTML behind the web form. Remember how the Google text input field had an attribute of name="q", which corresponded with the parameter q= in the URL? The name= attributes in other forms similarly comprise the keys that need to be packed into the params= dictionary.

So in the HTML snippet above, find every single tag that has a name= attribute. One text input element has lstinlinename="lname" and the other has name="fname". The only other fields that have name attributes are the two buttons. Notice also that the button you click to launch the search has name="submit" and a second attribute called value="Get Listing".

Given all of this, we might try the following parameters dictionary: parameters = {'lname':last, 'fname': first , 'submit':'Get Listing'}

As it happens, this works! The following code creates a simple command-line tool to simply print the resulting HTML with the query results. Processing this data is, once again, left as an exercise for the reader.

Note to 2018 students: As you know, I'm writing this just days before I distribute it. The code snippet below worked in 2017 but is giving me an error in 2018. Because I know that the general approach is sound, I'm including the code. But you might encounter errors if you try to run it. If you figure out what I did wrong, please let me know!

```
1  import requests
2
3  url = "http://docquery.fec.gov/cgi-bin/qind/"
4
5  last = input ("Last Name: ")
6  first = input ("First Name: ")
7
8  parameters = {'lname':last, 'fname': first , 'submit':'Get Listing'}
9
10 response = requests.post(url, data=parameters)
11
12 print(response.text)
```

16.6 This won't always work

Alas, some web pages cannot be scraped, at least not with the tools and techniques we've presented here. There are two major design features that make a

webpage hard to scrape, and sadly for us scrapers, these features are proliferating around the web in 2018.

16.6.1 Javascript

First, many web pages take advantage of the fact that every modern browser has a full-blown computer programming language interpreter at its core. The language is Javascript, and because it comes built-in to browsers, websites increasingly send Javascript along with HTML. In fact, on many web pages, almost all of the code is Javascript and not HTML.

Web pages embrace Javascript because, unlike HTML, it is a full programming language, with while loops and if statements, and variables, and lists. This means that a Javascript-enhanced web page can be far more interactive and dynamic than one that uses boring, old HTML.

Unfortunately, Beautiful Soup won't parse Javascript. Even worse, it's far more difficult to make sense out of what Javascript is doing than it is with HTML.

There are other libraries that work around the Javascript problem. One popular one is Selenium. This is outside the scope of the book, but you can find lots of good online help about it.

16.6.2 Countermeasures

Second, many web pages proactively discourage web scraping, and some even deploy technical countermeasures to prevent it. One classic method is to see if the browser visiting the site knows how to interpret Javascript. If not, it's probably a computer program rather than a human with a browser, and the web server can dynamically change the HTML that is returned to omit the information to be scraped. There are other approaches, such as examining what is known as the "user-agent string", which your browser's self-disclosure revealing the type and version number of your software.

All of these countermeasures can be defeated with counter-counter measures, but you see how this all leads to an escalating arms race. The details of this race are also beyond the scope of this book.

Chapter 17

APIs

This is a *very* rough draft of this chapter. You won't find nearly as much context as you have in other chapters. You'll also spot a lot of rough prose and possibly even some errors in the examples. Please bear with us! And please let us know if you spot any errors.

We haven't yet written a cheatsheet for this chapter. We hope that the spartan and straightforward nature of this draft makes a cheatsheet a little less necessary.

17.1 What is an API?

Generally speaking, an "application program interface", or API, is a means for transferring data between two computers across a network (or between two programs running on the same computer). The term usually describes both the technological mechanism for transferring the data, but just as importantly, the fully-specified "interface" that tells the person using the API how to create a properly formatted request for the data, and what format the data will be in when it arrives.

Specifically, in this chapter, we're talking about APIs that use the http protocol–the protocol that drives the web–as the method for data transfer. These kind of web APIs have become an increasingly important source for data of all kinds online. Entities large and small now give free access to the public of the data they maintain through APIs.

For lawyers, many sources of data that have long been useful for legal practice can now be obtained via an API. Many government agencies, for example, provide useful APIs to access their records. The U.S. Patent and Trademark Office gives docket information relating to its Patent Trials and Appeals Board using an API; scientific agencies make data about climate or oil production or pollutants available through APIs.

You already know just about all the Python you need to know to use APIs. Just as the requests library is useful for downloading HTML-formatted web

pages or PDF documents, it can also be used to download information using an API. The only difference is you visit a URL that provides an API rather than provides a web page. And many APIs, including the ones we're going to highlight, transmit data using JSON, so you can use the json library to transfer data between JSON and Python data structures.

APIs thus pose a bit of a challenge for a programming book author. Since all we want to do is teach you how to use APIs, generally, this can be a very short chapter. You already know most of what you need to know. The point of this chapter isn't to help you master any particular API, although we'll use a few examples along the way.

Given this challenge, we hope to accomplish three goals in this chapter, which are different from the "teach more Python" goals of past chapters. These goals are:

1. Unfortunately, API designers tend to be a jargon-loving bunch, and they've given new names to techniques and concepts that you've already learned about under a different name. So much of this chapter will be about translation.

2. Through our examples, we'll give you some strategies for figuring out how to use APIs.

3. We'll include a specific discussion about authentication. You'll often need to register for a (usually free) account before you can use an API, and it's something a bit confusing to figure out how to register your account from within Python.

17.1.1 APIs versus Web Scraping

The last chapter spoke a lot about turning unstructured web data into structured data. The designer of a web page is primarily publishing information intended for human consumption, meaning it privileges visual design and aesthetics over separating data into usable chunks. The key challenge for the web scraper is to write a program that can look for these visual design features, in the form of HTML tags, to delineate the separator points between bits of interesting and useful data.

In contrast, an API is usually fully-structured, meaning it is already delimited into pieces that are labeled and (usually) well-documented.

Given all the advantages an API has over web scraping, why web scrape? Because not all useful data has already been turned into an API. Even in 2018, plenty of really useful data is published only in HTML, meaning the only way to unlock it is to scrape it.

There's another slightly more philosophical answer: APIs are highly regulated by the data producer. They reveal only the information the publisher wants to reveal, in exactly the format it wants to publish it. The web is far less regimented and "enclosed". It's "permissionless" in a way that an API is not. You can't use an API until the data owner decides to give you permission; you can scrape data as soon as it is published on the web.

17.2 APIs and Python

The APIs we're interested in use the same-exact http protocol we introduced you to in the last chapter as the core technology of the web. Even so, the line between web and API is pretty blurry. We tend to think of the web as the communications medium that uses http but also that involves the transfer of HTML meant to be viewed by humans in browsers. APIs, in contrast, tend to be written to be accessed by a computer program, aren't typically meant to be viewed in a browser, and tend to speak JSON (or another format we won't discuss called XML).

To summarize these differences in a table:

	web	API
protocol	http	http
designed for	humans	programs
tool used to interact	browser	Python program
return format	HTML	JSON

The great news it that the Python requests library can speak to both HTML-based web pages and JSON-based APIs.

17.2.1 A simple API

To demonstrate how to use a simple (but useful) API from Python, we'll focus on a non-legal example: the API hosted at the URL: http://api.citybik.es. Visit this URL from a web browser:

<screen shot>

CityBikes provides an API for querying the status of bike sharing systems from around the world. (As an interesting aside, this API gets its data from a project called pybikes, which is a Python library that provides dozens of purpose-built web scrapers for different bike sharing websites!)

Even though APIs are intended to be used programmatically (from a computer program), not interactively (from a web browser), all good APIs provide a base URL that, when visited in a browser, provides documentation on how to use the API. In a browser, http://api.ciybike.es displays the documentation for this URL.

This documentation explains that this API provides two "Endpoints". For example, one of those endpoints is http://api.citybik.es/v2/networks. Visit that URL in your browser, and you'll find a very long list of what appears to be JSON-formatted ata, listing a little information about dozens of city bike systems from around the world. Here's an excerpt of the listing:

```
{
  "networks": [
    {
      "company": [
        "Bike U Sp. z o.o."
```

```
      ],
      "href": "/v2/networks/bbbike",
      "id": "bbbike",
      "location": {
        "city": "Bielsko-Bia\u0142a",
        "country": "PL",
        "latitude": 49.8225,
        "longitude": 19.044444
      },
      "name": "BBBike"
    },
    {
      "company": [
        "PBSC",
        "Alta Bicycle Share, Inc"
      ],
      "href": "/v2/networks/melbourne-bike-share",
      "id": "melbourne-bike-share",
      "location": {
        "city": "Melbourne",
        "country": "AU",
        "latitude": -37.814107,
        "longitude": 144.96328
      },
      "name": "Melbourne Bike Share"
    }
    ...
}
```


What exactly is an *endpoint*? There are at least two useful answers to the question: First, an endpoint is an instruction for obtaining useful information from the API. In this sense, endpoints are a little like Python functions or methods–when you "call" the endpoint, it returns JSON data. (In fact, there are endpoints for complicated APIs that, just like Python functions and methods, do much more than just return data–endpoints that upload data or do something useful, but the APIs we'll review are simpler and do little more than return information.)

The second useful way to think about endpoints is that they're just URLs. We've already mentioned that the API community has chosen to rename old, familiar concepts, perhaps for good reason, but in ways that might be confusing for new programmers. Endpoints are usually displayed as what relative URLs, although on this page, they are listed as absolute URLs.

So what exactly does the endpoint in this example provide? The documentation doesn't say, but with such a simple API, you can study the examples and probably guess: This endpoint appears to return JSON giving a little bit of information about every bike system it has data about.

This JSON returns a dictionary with a single key, "networks", which points to a list of dictionaries, one for each bike system. For example, here is a snippet of the full JSON data, the dictionary describing the "Capital Bikeshare" system in DC:


```
{
  "company": [
    "Motivate International, Inc.",
    "PBSC Urban Solutions"
  ],
  "gbfs_href": "https://gbfs.capitalbikeshare.com/gbfs/gbfs.json",
  "href": "/v2/networks/capital-bikeshare",
  "id": "capital-bikeshare",
  "location": {
    "city": "Washington, DC",
    "country": "US",
    "latitude": 38.8967584,
    "longitude": -77.03701629999999
  },
  "name": "Capital BikeShare"
},
```


Notice the "href" key. It points to what looks like a relative URL, /v2/networks/capital-bikeshare. As you might've guessed, this is another endpoint for this API. In fact, it's the only other kind of endpoint this API offers, every bike system has a unique ID (in this case capital-bikeshare) that can be appended to /v2/networks/ to generate an endpoint to learn more about that system.

Once again, remaining in the browser, let's build a URL out of this "relative URL", ahem, endpoint. The resulting URL is http://api.citybik.es/v2/networks/capital-bikeshare. Visiting this in your browser returns a much larger amount of JSON.

This JSON resolves to a dictionary with a single key, "network". This points to a dictionary that contains many of the same entries listed above. But there is an additional key called "stations" which points to dozens (maybe more than a hundred) separate dictionaries. Here are two, as they appeared at the time of writing:


```
{
    "empty_slots": 3,
    "extra": {
      "address": null,
      "last_updated": 1522842197,
      "renting": 1,
      "returning": 1,
      "uid": "430"
    },
```

```
    "free_bikes": 10,
    "id": "7b35012cb8e0b6530228e4cca3b71a56",
    "latitude": 38.931911,
    "longitude": -77.219261,
    "name": "Jones Branch & Westbranch Dr",
    "timestamp": "2018-04-05T01:22:14.191000Z"
  },
  {
    "empty_slots": 13,
    "extra": {
      "address": null,
      "last_updated": 1522887467,
      "renting": 1,
      "returning": 1,
      "uid": "473"
    },
    "free_bikes": 5,
    "id": "7e5dfbd097bc57ca0d02e4e329353d0a",
    "latitude": 38.905368,
    "longitude": -77.065149,
    "name": "Potomac & M St NW",
    "timestamp": "2018-04-05T01:22:14.622000Z"
  },
```


As you can probably tell, these two dictionaries reveal the current status of two bike stations in Washington, DC, including recently-updated information about the number of bikes available at this moment, 10 at the Jones Branch location, and 5 at the Potomac St station.

As we have said, for some reason, the API development community has chosen to use new labels to refer to web concepts you have already encountered. Here are three terms in particular that you can understand as essentially the same thing as something you've learned before:

API term	Web term
Resource	web page
URI	URL
Endpoint	Relative URL

17.2.2 The primitives of an API: requests and json

You already know all of the Python you need to know to use many APIs. Most APIs are simply built on GET or POST queries, meaning you use `requests.get()` or `requests.post()` to access.

Many APIs speak JSON as their native tongue. They accept data structures formatted in JSON as user input, and they return data structures formatted in JSON to send output back to the user. Now is a good time to re-acquaint yourself with Chapter 11.4 (JSON) and Chapter 17 (requests).

Let's write code that will list all of the City Bike stations in Washington, D.C., printing for each one the number of bikes available at the moment. Unfortunately, the API documentation doesn't say whether it responds to GET or POST queries, but when in doubt, it's good to try GET first:

```
1  import requests
2  import json
3  import urllib
4
5  endpoint = "/v2/networks/capital-bikeshare"
6  base_url = "https://api.citybik.es"
7  url = urllib.parse.urljoin(base_url, endpoint)
8
9  response = requests.get(url)
10 data = json.loads(response.text)
11 stations = data["network"]["stations"]
12
13 for station in stations:
14     print("{} bikes available at {}".format(station["free_bikes"], station["
           name"]))
```

This works! At the moment this was written, here were the first few lines printed, out of dozens:

```
$ python3 bikes.py
10 bikes available at Jones Branch & Westbranch Dr
10 bikes available at Town Center Pkwy & Bowman Towne Dr
5 bikes available at Potomac & M St NW
0 bikes available at 22nd St & Constitution Ave NW
3 bikes available at 18th & Eads St.
10 bikes available at 15th & Crystal Dr
4 bikes available at Aurora Hills Community Ctr/18th & Hayes St
14 bikes available at Pentagon City Metro / 12th & S Hayes St
...
```

Notice that you could understand this code in the previous chapter. There isn't really any new Python here. The only potential novelties here are the fact that this is the first time we've used response.text, in line 10, to return anything except HTML; and the fact that we called the variable in line 5 "endpoint" rather than "relative_url".

Notice also the progression of the past few chapters. This code can pull fully structured Python data structures from the JSON, which is much more straightforward and less error-prone than using Beautiful Soup to extract data from between HTML tags. And Beautiful Soup was, in turn, much simpler than hunting through raw data using regular expressions.

This is the power of APIs: you can use ordinary and simple web downloading code to access really useful data.

17.2.3 A little more on POST queries and the requests library

In the last chapter, in section 17.5.6, we introduced you to POST queries, but because many APIs use POST queries, let's say a little more about them now. (Note to 2018 students: this probably deserves to be in the last chapter, but for your benefit, I'm just adding it here.)

Up until now, most of the websites we have looked at have used GET queries, which means that data sent from the user to the webserver has been transmitted via a URL query string, either through munged URLs or through the params= argument to the requests.get() function.

Consider some of the shortcomings of the GET method. First, what if you want to send a lot of data to a web server? What if, for example, you wanted to send an image, or a document, or just paragraphs full of text? Second, how do you send hard-to-print characters, such as carriage returns, tabs, and special characters using GET?

The answer is what is known as POST, the other way to send information to a web server. POST queries don't send data via query strings in URLs. Instead, they send the data through a different channel, one which doesn't leave behind visible trails of information in the browser, the way a GET query does in the URL bar.

With Python's requests library, you send a POST query by replacing requests.get() with, you guessed it, requests.post(). The basic operation is identical: you pass a URL and any optional arguments, and the function returns a response object that is identical to the response objects returned by requests.get().

The primary difference is in how you pass your data to the URL you are visiting. URL munging obviously won't work, so you can't just manipulate the string you pass as the URL. Instead, there are several possibilities that vary based on what the website you are visiting is expecting. In many cases, the web server will just expect a dictionary of information, with keys specified by the website and values full of your information. This is essentially identical to using a GET query with the params= argument, but POST queries want that data to use a different argument for that dictionary: data=. That's the only difference.

Unfortunately, some websites expect JSON-encoded dictionaries for their POST queries instead. While you could just import json and json.dumps() your dictionaries, requests provides an easier way. Instead of using data= for your argument, use json= instead.

Bottom-Line: if you're sending data using a POST query rather than a GET query, and you have a dictionary in the variable d full of keys and values formatted according to that website's wishes, then use either requests.post(data=d) or requests.post(json=d) to send that data.

And how do you know whether to use data= or json=? In an ideal world, the website you're using will have crystal clear documentation that will tell you which to use. In the world we actually occupy, one in which great documentation is sometimes hard to find, you might need to use trial and error, so if data=

doesn't work, try json= instead.

17.2.4 Sending data to an API

Back to APIs. Let's switch to a more complicated API, one a bit more closely related to legal applications. In 2014, President Obama signed the DATA Act into law. This law requires the federal government to make data about federal expenditures more transparent and easier to access. This system is supposed to track all grants, contracts, and loans, issued by government agencies, and to make that data easy-to-access. It accomplishes this, in part, by mandating the creation of an API.

The API that resulted is available at https://api.usaspending.gov/. We're writing this just one week after version 2 (v2) of the API was launched. No doubt some of the details will change by the time this book ends up in print, so consult this URL in your browser to see what is new.

This is a pretty complex API. About a dozen endpoints are listed at the documentation at https://api.usaspending.gov/docs/endpoints.

Let's focus on just one endpoint, https://api.usaspending.gov/api/v2/bulk_download/list_agencies/, which is documented at https://github.com/fedspendingtransparency/usaspending-api/blob/stg/usaspending_api/api_docs/api_documentation/download/list_agencies.md.

As of the date of this writing, the documentation for this endpoint begins:

List Agencies
Route: /api/v2/bulk_download/list_agencies/

Method: POST

This route lists all the agencies and the subagencies or federal accounts associated under specific agencies.

Request example
{
"agency": 14
}
Request Parameters Description
agency − agency database id. If provided, populates sub_agencies and federal_accounts.

Notice that they've introduced more jargon! "Endpoints" are also known as "Routes". Notice also that this endpoint uses the POST rather than GET method, so you must use requests.post() rather than requests.get() in your code.

Let's try calling this API as simply as we know how:

```
1 import requests
2 url = "https://api.usaspending.gov/api/v2/bulk_download/list_agencies/"
```

```
3 response = requests.post(url)
4 print(response.text)
```

This prints a large amount of JSON, beginning with:

```
{"sub_agencies":[],"agencies":{"cfo_agencies":[{"toptier_agency_id":14,"
    cgac_code":"012","name":"Department of Agriculture"},{"
    toptier_agency_id":15,"cgac_code":"013","name":"Department of
    Commerce"},{"toptier_agency_id":126,"cgac_code":"097","name":"
    Department of Defense"},
...
},"federal_accounts":[]}
```

Just as the documentation promised, this is a list of all of the "top-tier agencies" that has data about its expenditures retrievable with this API.

What else can we do with this endpoint? Look again at the documentation. Notice that this endpoint accepts an optional argument called "agency". Earlier, we compared API endpoints to Python object methods. Just like a method, many APIs take arguments that can be used to return different pieces of information from the underlying database.

How are arguments passed to an endpoint? Again, you can build what you've learned before. For endpoints that use the GET method, you tend to pass arguments using either URL munging or the params= argument to the requests.get() function. For endpoints that use the POST method, like this one, you tend to pass arguments using the data= argument or json= argument to the requests.post() function.

In this API, the "agency" argument takes an integer, the ID number that this system uses to refer to a particular agency. Those are exactly the integers that appear in the output listed immediately above under the dictionary key toptier_agency_id. For example, the "Department of Agriculture" appears to have an ID of 14, the "Department of Commerce" is 13, and the "Department of Defense" is 126.

So let's modify our code to pass learn more about the Department of Defense:

```
1 import requests
2 url = "https://api.usaspending.gov/api/v2/bulk_download/list_agencies/"
3 args = { "agency": 126 }
4 response = requests.post(url, data=args)
5 print(response.text)
```

This doesn't find anything. What went wrong?

```
{"detail":"Agency ID not found"}
```

This is one of those situations where the API wasn't expecting data= data. It was expecting JSON input. Changing the name of the optional parameter is all you need to do to fix this:

```
1 import requests
2 url = "https://api.usaspending.gov/api/v2/bulk_download/list_agencies/"
```

```
3 args = { "agency": 126 }
4 response = requests.post(url, json=args)
5 print(response.text)
```

This prints a large amount of JSON data, listing every sub-agency of DoD in the expenses database. Here's a snippet:

```
{"sub_agencies":[{"subtier_agency_name":"Army Corps of Engineers Civil
    Works"},{"subtier_agency_name":"Defense Advanced Research Projects
    Agency"},{"subtier_agency_name":"Defense Commissary Agency"},{"
    subtier_agency_name":"Defense Contract Audit Agency"},{"
    subtier_agency_name":"Defense Contract Management Agency"},{"
    subtier_agency_name":"Defense Finance and Accounting Service"},{"
    subtier_agency_name":"Defense Health Agency"},{"
    subtier_agency_name":"Defense Human Resources Activity"},{"
    subtier_agency_name":"Defense Information Systems Agency"},{"
    subtier_agency_name":"Defense Intelligence Agency"},
...
}
```

17.3 Authentication

An API is an invitation to the world to use the endpoints specified with the parameters specified to grab data of potential interest. But, like web developers who deploy countermeasures against scraping, API providers still worry about abuse and overuse. They want to prevent people from downloading too much data or downloading data for certain disfavored uses.

To try to combat these problems, many API creators add a thin layer of accountability by restricting the use of their APIs to users with pre-registered accounts. Before you can use such an API, you have to visit a website and create a user account, just like you would on any other website.

The big difference from user accounts you've created on the web is that because this is designed to be used in code, not directly by a human, it doesn't make sense to give you a traditional username and password. Often, API registration systems will just give you the password, which it generates, which is traditionally known as a "key" or "API key". As the last step in registration, they'll display the key in your browser or maybe send it to you via email. From then on, every time you use the API, you send the API key back as part of the data you submit with your `requests.get()` or `requests.post()`, perhaps as an element in the `params=` dictionary.

Increasingly, API providers are switching to more industrial-grade authentication schemes that you can use to access different providers using a single key. All of the major platforms (Google, Facebook, Twitter, etc), for example, provide authentication that you can use throughout their varied services, but also with related third-parties.

Speaking of platform providers, think of one other reason why you might want to authenticate. Up until now, all of our examples have involved access to public data. But APIs are also useful for accessing your own personal data. For example, Google publishes an API that lets you interact with your Gmail inbox, and Facebook provides an API for interacting with your Facebook account.

These APIs are the way that bots operate. Twitter bots, for example, are simply constantly running computer programs that have authenticated access (probably through an API key) to a particular Twitter account (or 10,000). Using the Twitter API, these bots can tweet messages and respond to replies.

As a general rule of thumb, the more industrial-strength the authentication you are trying to access, the more complex and burdensome the authentication setup will be. On simple APIs, authentication is just as easy as setting up a web account, and "logging in" is as simple as putting the right API key into the right parameter entry. On the other hand, using Google or Twitter's API system tends to require many more steps and many more lines of code.

Chapter 18

Machine Learning

We now live in the age of machine learning, the technology that powers spam filters, image classifiers, automated translators, smart assistants (like Siri and Alexa), self-driving cars, and a multitude of other recently-released or soon-to-be-released products. This is only the beginning. The techniques behind contemporary machine learning are still in their infancy, and we have only just begun to understand how to use them to their full potential.

Perhaps to your surprise, the most popular tools for performing machine learning are written in Python. In this chapter, we will teach you basic proficiency with two libraries, `scikit-learn` and `keras`, that are used to develop industrial-strength machine learning models. Although machine learning has its own mystique above and beyond the mere act of programming, it is entirely within the reach of any reader who has made it to this point in the book.

In this chapter, we will discuss the conceptual principles of machine learning (Section 18.1) before concretely presenting two machine learning techniques in particular: logistic regression (Section 18.2) and neural networks (Section 18.3).

Logistic regression is a standard statistical technique that allows you to predict whether a piece of data is a member of one of two categories. For example, one could use logistic regression to predict whether a picture contains a cat or a dog or whether a Supreme Court justice is likely to vote for or against one side of a case.

Neural networks, whose name suggests yet further mystique, are really just a further refinement of logistic regression. (They have nothing to do with the human brain, regardless of what the name may suggest.) However, this refinement gives neural networks dramatically more expressive power. Neural networks power nearly all modern machine learning applications: personal assistants, automated translation, image classification, and self-driving cars. Not only is the topic of neural networks vast, but it is evolving quickly. In Section 18.3, we cover fundamentals that have withstood the test of time, giving you a jumping off point for further learning.

Entire books have been written on machine learning, and we cannot convey all of that knowledge in a single chapter. Instead, our goal is to give you

enough information so that you are capable of reasoning fluently about machine learning, making basic use of two specific techniques, and continuing to learn on your own.

Reassurance for the mathematically inhibited. As you are no doubt aware by this point in this book, programming does involve a small amount of elementary math. Although we have labored to keep these mathematical intrusions to a minimum, numbers have surfaced from time to time. As we acknowledged at the beginning of this book, lawyers tend to have an unfounded, yet pathological, aversion to math. If you have made it this far, we congratulate you for having overcome those inhibitions.

We repeat and emphasize the same advice here. Machine learning is based on statistical machinery that will require you to do some amount of interacting with numbers. The mathematical concepts necessary for this chapter are slightly more advanced; however, we assure you that you learned all of the tools you need for this chapter in high school, and we will refresh your memory in the few moments where doing so is necessary. Underpinning both logistic regression and neural networks is a small amount of calculus, but the libraries we use hide this away completely. Just as you have done throughout this book so far, we ask that you trust us to guide you through this math unharmed.

18.1 Principles of Machine Learning

What, exactly, is machine learning? We define machine learning using two criteria:

1. Machine learning involves developing a model—a simplified version—of some process (e.g., predicting how a Supreme Court justice will vote on a particular case). This model should have predictive power (i.e., it should be at least somewhat accurate).

2. This model is developed by learning from existing data (e.g., how a justice has voted in previous cases) in order to make predictions about new data (e.g., how a justice will vote in the future).

In this section, we will sharpen our understanding of both of these qualities and discuss the way engineers set up and solve machine learning problems. At its heart, machine learning is essentially automated pattern recognition. It aims to extract a pattern from existing data that is useful for making sense of new data.

Where does the term artificial intelligence (which we have carefully avoided to this point) fit into this picture? Artificial intelligence is a vast, amorphous, and ambiguous term that captures any computer-aided reasoning. Machine learning is certainly a kind of artificial intelligence. The challenge with the term "artificial intelligence" is that, depending on how it is defined, any computer program can fall within its purview. Does the program we wrote assessing

whether a user is eligible for a public benefit program qualify as artificial intelligence? It's hard to say. Does it qualify as machine learning? Not according to our definition: there is no process of learning from data to develop a model.

Machine learning is currently the dominant way of achieving artificial intelligence, and it provides a well-developed framework for situating problems that we hope to solve with artificial intelligence. All of the concepts and techniques in ths chapter are specific to machine learning.

18.1.1 Models

A model is a simplified (often oversimplified) way of summarizing a much more complicated process. Models typically aim to make predictions about how changes in circumstances will lead to changes in outcomes. For example, an economic model might aim to predict how a tax cut (i.e., a change in economic circumstances) will affect economic growth, unemployment, and future tax revenues for the government (i.e., changes in outcomes).

By definition, models are not correct. By simplifying the world, a model will inherently sacrifice some degree of accuracy. However, it is often productive to sacrifice a small amount of accuracy to achieve a vastly simpler model. As a famous statement by statistician George Box goes, "All models are wrong, but some models are useful."

Types of machine learning tasks. Most machine learning tasks can be grouped into two categories: classification and regression. Classification tasks involve predicting whether a particular input falls into one of several categories. For example, we might wish to predict whether a particular Supreme Court case will be decided in favor of the petitioner (a category) or in favor of the respondent (another category). Likewise, one might wish to determine whether an FCC comment was favorable to a rule (a category) or unfavorable (another category). Finally, we might wish to analyze writing style to determine which of nine possible Supreme Court justices (nine different categories) wrote a particular opinion. In practice, classification models typically predict the probability (i.e., the percent chance) that an input falls into a particular category rather than making a clear decision to pick one category over the others. We will explore this idea later in the chapter.

In regression, we try to predict a specific output for each input. For example, a model might aim to predict the change in GDP attributable to a change in tax policy. The model predicts a concrete number, for example that GDP will increase by 0.3%. As another example, a model might try to predict the number of representatives who will vote for a particular bill.

In this chapter, we will mainly consider models that perform classification rather than regression, but it is important to keep in mind that regression is an equally natural form of machine learning model.

Parameterizable models. In most machine learning contexts, we usually consider models that are parameterizable. To understand this concept, consider

an example: we want to develop a model that predicts which side a Supreme Court justice will vote on a particular case based on the number of questions she asks both sides. Initially, our model might predict that she will vote against the side to whom she asked more questions.

However, we may have to change this model. Perhaps justices are reluctant to reverse the decisions of lower courts. It might be that a better model predicts that the justice will give some degree of deference to the side that won in the lower court even if she asks it slightly more questions. Concretely, one possible model is that the justice will vote for the side that won in the lower court unless she asked that side, say, five more questions than she asked the side that lost in the lower court. In other words, asking five more questions of the side that won in lower court overcomes any deference the justice might previously have granted. The number five is arbitrary. The correct number might be two, seven, twelve, or even zero (i.e., no deference). The number of questions is a parameter that we can change about our model depending on what we learn from looking at real data—previous Supreme Court transcripts.

In machine learning, we usually begin by selecting a model family—a model with one or more parameters that can be changed to create a wide variety of different models. In the preceding example, the model family considers the difference between the number of questions a justice asked to the two sides in a Supreme Court case. For example, the justice asked one side five questions more than the other. It has a single parameter: the threshold at which this difference overcomes the deference due to a lower court's decision.

18.1.2 Learning

The other distinguishing quality of machine learning is that the parameters for a model are selected by examining data about which we already know the answers. For example, to develop a model that predicts how a justice will vote on future cases, we consider how she voted on previous cases. The act of tuning the parameters of a model based on pre-existing data whose answers are known ahead of time is known as learning or training. The pre-existing data itself is known as training data. The act of using a model to make predictions about new data is known as inference.

The typical workflow for creating and using a machine learning model is as follows:

1. Select a particular model family that we wish to train.
2. Acquire some training data that includes both the inputs that we intend to provide to our model (e.g., transcripts of Supreme Court cases) and the actual outcomes (e.g., how a justice voted).
3. Use the training data to find the parameters that make the model as accurate as possible.
4. As necessary, supply new input data (whose answers are unknown) to the model and ask it to make predictions about what the answers will be.

It is important to note that training is typically performed in its entirety before the model is ever used to make predictions. After the model is deployed for inference, no further training ever occurs. There are occasionally some situations in which a model is intermittently trained, utilized, and further re-trained, but they are relatively rare in practice.

Data and features. How, exactly, do we turn complicated information (like the transcript of a Supreme Court oral argument) into concise data on which a model can operate (whether performing training or inference)? Before providing data to a model, we typically break it down into smaller pieces that summarize interesting aspects of the data. For example, the number of questions a justice asked to each side in a case. Something is clearly lost when simplifying data down from the raw transcript of an oral argument to the number of questions that were asked. However, the number of questions (two integers) is much easier to mathematically process than the transcript (a string containing many words spoken by many participants) and may distill important information that a model would have a hard time extracting on its own.

If we wanted to, we could provide the model with many such data points about each case, such as the amount of time that the justice spent speaking and whether each question was supportive or critical of the particular side's case. Each of these data points is known as a feature, and the process of turning raw data into a small set of features usable by a model is known as feature extraction. Feature extraction is very closely related to the process of data cleaning, in which raw data is processed into a more regular, structured form amenable to analysis.

Training, testing, and validation. Even after we have performed feature-extraction on our training data to convert each entry from raw information into usable features, we must still perform one further step before we are ready to train our model. Specifically, we randomly divide this data into three parts which we call training data, validation data, and testing data. We will train the model using the training data and evaluate how well it performs using the validation and test data.

To understand why it is essential that we perform this extra step, consider a motivating problem: once we have trained our model, how do we know how well it performs? At first glance, one idea is to see how accurately it makes the right predictions on the training data. After all, we want our model to perform as accurately as possible, so this is a reasonable way of measuring success. The problem with this approach is that our model can perform exceedingly accurately by simply memorizing the training data. If it does so, it will only work on the training data and fail to generalize to new data, meaning its performance will be terrible on new data. The act of memorizing the training data at the cost of generalization is known as overfitting. Overfitting to training data is a persistent problem in machine learning, even for experienced professionals.

In order to detect and avoid overfitting, we make use of the validation data

and test data that we set aside at the beginning of this subsection. We train the model parameters on the training data and evaluate the performance of the model on the validation data. Since the model has never seen the validation data before, trying the model on the validation data gives us a way of detecting overfitting and changing our training procedure to prevent it from happening.

Unfortunately, we can sometimes accidentally change the training procedure so many times that we overfit our training procedure to the validation data. In order to detect this kind of overfitting, we assess the final accuracy of the model on the test data. Since the model does not see the test data until we are completely done developing the model, we can be assured that the test data gives us an accurate portrayal of the model's performance.

Optimization and loss. When we train a model, what is our goal? Training is a form of optimization in which we try to find the parameters that give our model the best possible performance. But how do we define the "best possible performance?" Doing so varies widely from problem to problem. Typical choices for regression and classification are completely different, and there is a wide range of typical choices. We typically try to develop a formula that characterizes how "incorrect" our model is; this formula is known as loss. Training involves finding the parameters for which our model reaches the minimum possible loss.

For classification tasks, loss is typically defined as the number of incorrect decisions that our model made on the training data. Recall that classification models don't usually output a clear decision that a particular piece of data falls into one category or the other. Instead, it will give the percent chance that it thinks a piece of data falls into each category. For example, a classifier might predict that there is a 70% chance that a justice will vote for the respondent in a particular case. A typical formula for loss calculates the difference between this percentage and the correct percentage. If the justice really did vote for the respondent, then the correct percentage was 100% and the loss is 30%. If the justice voted against the respondent, then the correct percentage was 0% and the loss is 70%.

In the sections that follow, we will discuss two specific machine learning model families and specific forumlas for loss that are popular in each of these settings.

18.2 Logistic Regression

Logistic regression is a standard statistical technique for performing classification. Given a set of training data, logistic regression learns to predict whether a set of features should be placed into one of two different categories. For example, given a set of pictures labelled as either "cat" or "dog," logistic regression could be trained to predict whether new, unlabelled images contain a cat or a dog. Later in this chapter, we will use logistic regression to build models of Supreme Court justices. Given a set of features about a case, can we predict whether a particular justice will vote for the petitioner or the respondent?

If you have taken a statistics class before, you are likely to have seen logistic regression. It is known under a variety of names (e.g., a logit model) in different fields. Logistic regression is a relatively simple machine learning technique. The underlying model family has only two parameters that are learned in the process of training a model. In contrast, neural networks often have millions of parameters. Even so, it is a very effective tool for a wide variety of classification problems.

Not only is logistic regression a valuable machine learning technique in its own right, but it is also the key building block used to create neural networks. Once you have a conceptual understanding of how logistic regression operates, neural networks are a natural next step.

In the first part of this section, we will introduce a concept that you probably learned in high school but have forgotten since: the notion of a mathematical function. The logistic function is a mathematical function that serves as our overall model family for logistic regression. In the second part of this section, we will introduce the logistic function and demonstrate how we can learn parameters for this model family to create useful machine learning models. Finally, in the third part of this section, we will use the `scikit-learn` library to build logistic regression models of Supreme Court justices in Python.

Libraries to install. In this section, we will make use of several heavy-duty numerical processing and statistical libraries for Python. These libraries are designed to perform machine learning and other numerical tasks as efficiently as possible. These libraries are quite large: they will each take up tens of megabytes of space on your hard drive and therefore may take a long time to download over a slow internet collection. To follow the examples in this chapter, you will need to install:

1. `numpy`, a library for efficient numerical processing.

   $ *pip3 install numpy*

2. `scipy`, a library for statistical analysis.

   $ *pip3 install scipy*

3. `scikit-learn`, a library for machine learning (including logistic regression).

   $ *pip3 install scikit-learn*

Throughout this chapter, we will display several charts that visually illustrate how the logistic function behaves. All of these charts are created using Python, and we will show you the commands that were used to create these charts. If you wish to follow along with these examples, you will need to install the `matplotlib` library:

$ *pip3 install matplotlib*

18.2.1 Mathematical Functions

The logistic function is a mathematical function that we will use to construct machine learning models. First, we will refresh your knowledge on what, exactly, a mathematical function is. You certainly learned about it in high school, but you may not have needed to apply that concept in the time since. A mathematical function is like any other Python function. It takes as input one or more numbers and returns a single number. For example, consider the following function:

```
1 def linear_function(x):
2     return 2 * x
```

At this point in your Python career, this function should look rather simple. It takes as input a number, `x`, and returns twice that number, `2 * x`. For example, `linear_function(2.3)` returns `4.6` and `linear_function(-1.7)` returns `-3.4`.

As you may recall from your high school days, we can represent this function visually. To do so, we will make use of the `matplotlib` library. We do not aim to teach the `matplotlib` library in this book, but you can optionally follow the examples in this book and experiment if you'd like.

```
1  import matplotlib.pyplot # A library for plotting graphs.
2  import numpy # A library for numerical processing.
3
4  # A function.
5  def linear_function(x):
6      return 2 * x
7
8  # Create a series of inputs ranging from -10 to 10.
9  x_values = numpy.arange(-10, 10, 0.1)
10
11 # Create the corresponding outputs of the function.
12 y_values = []
13 for x_value in x_values:
14     y_values.append(linear_function(x_value))
15
16 # Plot the values.
17 matplotlib.pyplot.plot(x_values, y_values)
18 matplotlib.pyplot.grid() # Show the grid.
19 matplotlib.pyplot.show() # Show the graph.
```

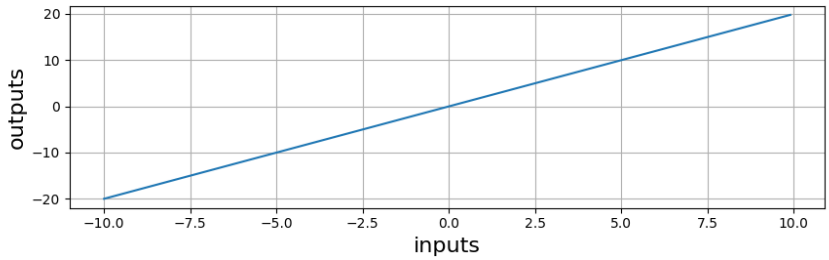
Before we show you the result of this code, we will briefly discuss what it does. The first two lines import the libraries we need to create this graph: `matplotlib.pyplot` and `numpy`. Lines 5 and 6 create the mathematical function we want to graph.

Now that we have the `linear_function` function, the rest of the program performs the steps necessary to graph it. On line 9, we create a list of inputs

for this function. To do so, we use the `numpy.arange` function. This function is identical to the `range` function we have used throughout this book: it creates a list of numbers starting at the first argument (`-10`), ending just before the second argument (`10`), and increasing by the third argument at each step (`0.1`). The difference between the `range` function and the `numpy.arange` function is that the latter can operate on floating point numbers. Strangely, the built-in Python `range` function cannot. Returning to the task at hand, line 9 creates a list of floating point numbers `[-10, -9.9, -.9.8, ..., 9.8, 9.9]` and saves it to the variable `x_values`.

Lines 12 to 14 build a parallel list containing the result of running each of these numbers through `linear_function`. In other words, `y_values` will store `[-20, -19.8, -19.6, ..., 19.6, 19.8]`.

Lines 17 to 19 use `matplotlib` to create the graph below.



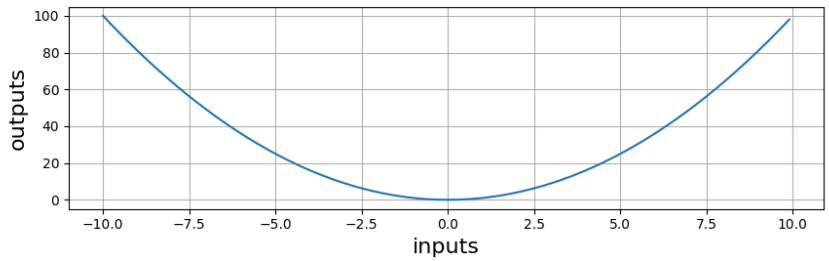
Let's take a closer look at this graph. Running left to right along the bottom of the graph are the values of the inputs that we provided to `linear_function`. Just as on line 9, the values range from -10 to 10. (As you may recall, the values running horizontally from left to right along the bottom are known as the `x-axis`.) The numbers running from bottom to top along the left of the graph are the outputs of `linear_function`. They range from `-20` to `20`, the range of outputs we produced and stored in `y_values`. (As you may recall, the values running vertically from bottom to top along the left are known as the y-axis.)

The blue line shows which input values correspond to which output values. For example, since the input value `-5` corresponds to the output value `-10`, the blue line passes through a point that is at `-5` on the x-axis (left to right) and `-10` on the y-axis (top to bottom). Likewise, since the input value `2.5` corresponds to the output value `5`, the blue line passes through a point that is at `2.5` on the x-axis and `5` on the y-axis. Since two times `0` is still `0`, the line passes through a point at `0` on the x-axis and `0` on the y-axis.

We can swap in other functions to get different graphs. For example, consider the graph produced by the function

```
def quadratic_function(x):
    return x * x
```

which squares its input (multiplies its input by itself).



The x-axis still ranges from `-10` to `10`, since those are still the inputs we provide to the function. However, the y-axis ranges from `0` to `100`. Rather than a line, we now get a curve that traces the relationship between inputs and outputs. Since `0 * 0` is still `0`, the input `0` on the x-axis corresponds to the output `0` on the y-axis. Both `5 * 5` and `-5 * -5` are `25`, so the inputs `-5` and `5` on the x-axis both correspond to the output `25` on the y-axis. Likewise, since `-10 * -10` and `10 * 10` are `100`, the inputs `-10` and `10` on the x-axis both correspond to `100` on the y-axis. The blue curve traces all of these points.

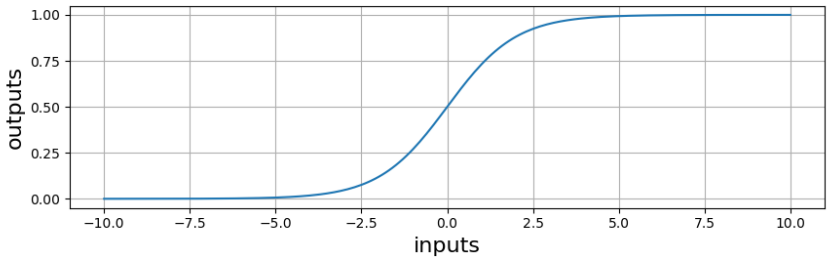
18.2.2 Logistic Regression in Concept

Now that you have refamiliarized yourself with mathematical functions and graphs, we will now introduce the logistic function, which we will use to build machine learning models. The logistic function is more complicated. You do not need to understand how this function works, and this is the first and last time in this book that you will need to think about this formula.

```
import math

def logistic_function(x):
    1.0 / (1 + math.e ** (-x))
```

To write the logistic function, we need to import the built-in Python `math` library to get access to the special value `math.e`, which is a mathematical constant like pi. Once again, you do not need to understand the math behind this function. The most important part of this function is its shape:



There are a couple of key facts about the logistic function that are useful for machine learning. First, notice that, when the input is very high, the output never gets bigger than `1`. In fact, no matter how high the input is, the output will never quite reach `1`. Likewise, no matter how low the input is, the output will never be smaller than `0`. In the middle of the graph, the output slowly switches over from being close to `0` to being close to `1` as the input grows from about `-5` to `5`. This switchover happens when the input is `0` and the output is `.5`, centered exactly between an output of `0` and an output of `1`.

How do we use this function as a machine learning model? The features that we provide to our machine learning model—a picture of an animal or data about a Supreme Court case—are the inputs to the logistic function. The output of the logistic function tells us its prediction. An output of `1` means that our model is 100% certain that the picture contains a cat or that the justice is going to vote for the petitioner. An output of `0` suggests that our model is 0% certain that the picture contains a cat or that the justice is going to vote for the petitioner, meaning, conversely, that it is 100% sure that the picture contains a dog or that the justice is going to vote for the respondent. Outputs in between `0` and `1` (e.g., `.3`) signify a certainty somewhere in between 0% and 100% (e.g., `30%`).

Consider the logistic function we have just graphed. This function represents a machine learning model that decides whether to place inputs into one of two categories which we'll abstractly call Category 0 and Category 1. In reality, these categories might be cat and dog or petitioner and respondent. Any input of `5` or greater falls into Category 1 with nearly 100% certainty and any input of `-5` or less falls into Category 0 with nearly 100% certainty since `logistic_function` outputs a clear `1` or `0`. Inputs in between are more uncertain. The input `-2.5` is 93% certain to fall into Category 0 since `logistic_function(-2.5)` is about `0.07`; similarly, the input `2.5` is 93% certain to fall into Category 1 since `logistic_function(2.5)` is about `0.93`. The model thinks that the input 0 has a 50/50 chance of being in either category, since `logistic_function(0)` is `0.5`.

Notice that this logistic regression model only works if the data actually fits exactly the right pattern. What if an input of `5` should still be in Category 0, and the model shouldn't switch over to Category 1 until the input is `10`? More concretely, consider a specific machine learning problem. Suppose we wish to model the chances that a law student will pass the bar exam based on her law school GPA. A typical GPA ranges from `0.0` to `4.0`, so the logistic regression

model we just outlined doesn't make any sense. We would expect that a student with a 4.0 GPA is almost certain to pass the bar (Category 1), but a student with a 2.0 GPA may struggle to do so (Category 0). The model should cross over from predicting Category 0 to Category 1 somewhere around a 3.0 GPA.

Consider a different machine learning problem. On the weekends, Larry the lawyer sells ice cream in a local park. He wants to determine whether today will be a profitable day (Category 1) or an unprofitable day (Category 0) to sell ice cream based on the temperature. We would expect that a temperature of 80 degrees is almost certain to be a profitable day to sell ice cream and a temperature of 20 degrees is almost certain to be an unprofitable day to sell ice cream. The model should cross over from predicting Category 0 to Category 1 somewhere around 50 degrees.

In order for logistic regression to capture both of these scenarios—bar passage rates and ice cream profits—it will need to work over a wide range of inputs. The graph of the logistic function we showed earlier captures input data that ranges from `-10` to `10` crossing over from one category to the other at `0`. The GPA example needs a model that can capture input data that ranges from `0.0` to `4.0` crossing over at `3.0`; for the ice cream example, the model must capture input data from `0` degrees to `100` degrees crossing over at about `50` degrees.

This is where machine learning comes in. Training a logistic regression model involves shifting and stretching the logistic function to fit the shape of the data. We can do so by modifying the logistic function slightly. Rather than computing

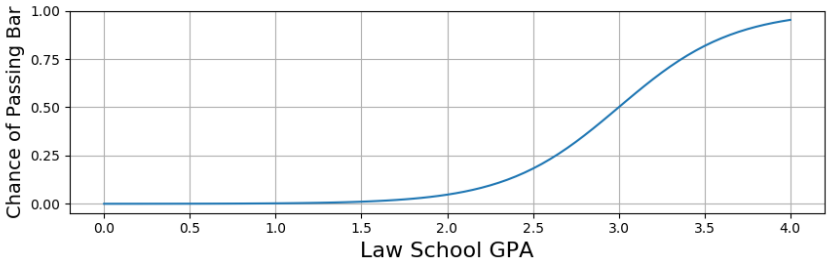
$$\text{logistic\_function}\,(\text{x})$$

we compute

$$\text{logistic\_function}\,(\text{w} * \text{x} + \text{b})$$

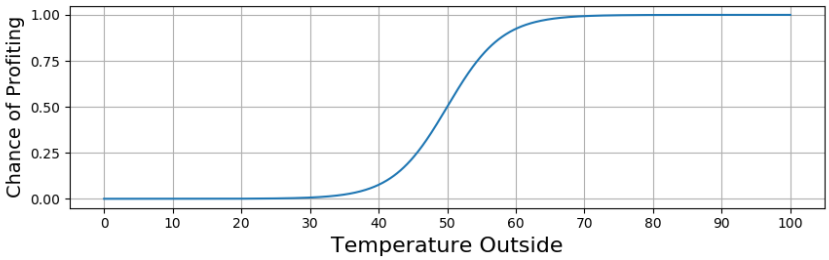
The variables `w` and `b` are the parameters of the logistic regression model. The `w` parameter controls how much the logistic function gets stretched or compressed in order to fit the range of the data. The `b` parameter controls how much the logistic function gets shifted to the left or right, changing where the crossover from Category 0 to Category 1 takes place. Learning from training data involves finding the right values of `w` and `b`.

For example, consider a logistic regression model for predicting whether a law student will pass the bar exam based on her law school GPA. By setting `w` to `3` (which compresses the logistic function) and setting `b` to `-9` (which shifts the crossover point to the right) we get the function `logistic_function(3 * x + 9)` whose graph is below:



This logistic function is a shifted and compacted version of the one we examined before. The function has been shifted to the right: it predicts that a student will have a 50% chance of passing the bar exam if she has earned a 3.0 GPA in law school. That is, the output reaches `0.5` when the input is `3.0`. In the original graph, this crossover point was instead at input `0`. The graph has also been compacted slightly. This model predicts that a student with a 1.0 GPA is almost certainly going to fail the bar exam, but that the chance of passing increases as the GPA does, reaching a 99% chance of passing when the student has a 4.0 GPA.

We could repeat a similar process for predicting ice cream sales. If we set `w` to `0.25`, which stretches the logistic function, and set `b` to `-12.5`, which shifts the function to the right, we get the function `logistic_function(.25 * x - 12.5)`, which has the following graph:



For temperatures of 30 degrees or less, this model predicts that there is a 0% chance that Larry will turn a profit (the logistic function outputs `0`). As the weather warms up into the 30s and 40s, the chance of profit increases. At 50 degrees, the chance of profiting reaches 50% (the logistic function outputs `0.5` on input `50`). As the temperature increases into the 50s and 60s, it becomes increasingly likely that Larry will profit. By the time it is 70 degrees outside, Larry is almost certain to profit (the logistic function outputs `1`). This model was created by changing `w` and `b` to shift the logistic function rightward (moving the crossover point to occur at 50 degrees rather than 0 degrees) and stretching the logistic function out so that it adequately covers a wide range of temperatures.

Summary. Let's consider logistic regression using our original framing of machine learning. We start with a model family: the logistic function. We then use training data to determine the exact parameters for the model. In the case of logistic regression, this involves setting `w` and `b` to the right values. We didn't discuss the exact algorithm for using the training data to find the best parameters because that is outside the scope of this book. The `scikit-learn` library, which we will use in the following section, does this automatically. Finally, once the model has been trained, we can predict how new inputs, like a GPA or a temperature, should be classified.

Multiple features. Notice that all of the examples we have considered so far have made predictions based on a single data point. For example, we predicted whether a law student would pass the bar based on her GPA alone. However, it is usually better to consider multiple features. We might also wish to take into account whether she took a bar prep class or the amount of time she has spent studying. Each of these features can be added to our logistic regression model:

$$\text{logistic\_function}\left(\text{w1} * \text{gpa} + \text{w2} * \text{prep} + \text{w3} * \text{studying} + \text{b}\right)$$

The variables `gpa`, `prep`, and `studying` are the features corresponding to the student's GPA, whether she took a prep class (`1` if she did and `0` if she didn't), and the number of hours she spent studying for the exam. Each of these features gets its own separate `w`. The model now has four parameters that have to be learned, `w1`, `w2`, `w3`, and `b`. This is a perfectly reasonable logistic regression model, and—since it has more data to work from—it is likely to be more accurate. However, this model is difficult to draw on a graph: since it has three separate inputs, it would require a four-dimensional graph to represent, well beyond the capabilities of the two-dimensional paper on which this book was written.

Most real-world logistic regression models have more than one (and often dozens) of features. For example, a logistic regression model operating on a picture of a cat might have hundreds of features, one for each pixel of the image. Although this point is subtle, it will be vitally important when we discuss neural networks.

18.2.3 Logistic Regression in Python (Scikit Learn)

In practice, logistic regression involves two major steps:

1. Assemble training and test data.

2. Train a logistic regression model.

As it turns out, the `scikit-learn` library makes the second step so easy that we will spent much of our time in this chapter focusing on the first step.

The Supreme Court Database. The Supreme Court Database is a collection of carefully labelled data about Supreme Court cases dating back to 1791. It includes detailed information about cases that the Supreme Court heard, how they arrived at the Supreme Court, the issues they covered, and how each justice voted. Our goal in this section will be to explore whether we can use logistic regression to develop a model of a justice's voting behavior that makes it possible to predict how she will vote in future cases.

The dataset we will use in this section can be obtained by visiting the Supreme Court Database at `http://scdb.wustl.edu` and downloading the MODERN Database, justice centered data, organized by Supreme Court citation, in CSV form. Download the file, unzip it to extract the CSV version of the database, and place it in the same folder as your Python programs. We are now ready to begin.

Before doing anything else, we need to import three libraries.

```
1  import sklearn.linear_model # Access sckit-learn's logistic library.
2  import csv # Process csv files.
3  import random
```

On line 1, we import scikit-learn's `linear_model` library, which contains the logstic regression library we will use later in this program. On line 2, we import the CSV library, which is necessary to load the Supreme Court Database into Python an extract its contents. On line 3, we import the `random` library, which will be useful for randomly shuffling our training data.

Now, we can import the database itself.

```
5  # Load the Supreme Court Database using a CSV DictReader.
6  scdb = csv.DictReader(open('scdb.csv'))
```

The first row of the CSV file contains the name of each column of the CSV file. This means that we can use a `csv.DictReader`, which will associate each value that it loads with the name of the column from which it came. Each row that this `csv.DictReader` loads will be in the form of a dictionary mapping a column name to the corresponding value in that row. Since this dataset has dozens of columns, it is valuable to be able to refer to them by name.

We can now begin assembling our training data. To do so, we create two lists.

```
8  # Variables to store the training data.
9  xs = []
10 ys = []
```

The first variable, `xs`, will store the inputs to the logistic function. The variable `xs` is a list of lists: each item that it stores is a list containing the features for one item. For example, one item in `xs` could be a list containing the lower court from which the case originated and the topic of the case. The

second variable, `ys`, will store, for each case, the justice's vote—the answer that we want our logistic regression to output when it sees the corresponding input. When the justice votes for the petitioner, the value in `ys` will be `1`; when the justice votes for the respondent, it will be `0`.

We are now ready to begin gathering training data. To do so, we will iterate over all of the rows in the `csv.DictReader`. Each row describes a single justice's vote. For now, we will try making a model of Antonin Scalia's voting pattern, but we could later substitute any justice.

```
12 # Assemble the training data.
13 for row in scdb:
14     # If this isn't a Scalia vote, skip this row.
15     if row['justiceName'] != 'AScalia': continue
```

On line 13, we iterate over each row in the `csv.DictReader` using a for-loop. Once inside the loop, the first action we take is to skip any rows that describe votes taken by any justice other than Scalia (line 14). The Supreme Court Database has a scheme for naming each piece of information about a row of the database. You can see the complete scheme at `http://scdb.wustl.edu/documentation.php`. The name of the justice who voted is under the key `'justiceName'`. On line 14, if we find that the justice is anyone other than Scalia, we immediately skip to the next iteration of the loop (and hence the next vote in the database).

Our next step is to determine the outcome: did Justice Scalia vote for the petitioner or the respondent? This step is a little tricky. The database does not tell us this information directly. Instead, it tells us whether Justice Scalia voted with the majority and whether the petitioner won. We will need to put these two pieces of information together to determine which way Scalia voted.

```
17     # Determine whether Scalia voted with the majority.
18     if row['majority'] == '': continue
19     majority = row['majority'] == '2' # True if he did.
20
21     # Determine whether the petitioner won.
22     if row['partyWinning'] == '2': continue
23     petitioner = row['partyWinning'] == '1' # True if so.
24
25     # Determine whether Scalia voted for the petitioner.
26     if majority and petitioner: y = 1
27     elif majority and not petitioner: y = 0
28     elif petitioner and not majority: y = 0
29     else: y = 1
```

Lines 17 to 29 determine whether Scalia voted for the petitioner (in which case the outcome we want from our classifier, `y`, should be `1`) or the respondent (in which ase `y` should be `0`). First, we use the `'majority'` column. This

column is sometimes empty, in which case line 18 skips that row. If it isn't empty, then a value of `'2'` means the justice voted with the majority and `'1'` means he voted with the minority. The boolean variable `majority` is `True` if he voted with the majority and `False` otherwise.

We follow a similar procedure on lines 22-23 to determine whether the petitioner won. If the `'partyWinning'` column is `'2'`, then the outcome was uncertain; in this case, line 22 skips this row. Otherwise, the column is `'1'` if the petitioner won and `'0'` otherwise. We save whether the petitioner won to the boolean variable `petitioner`.

Finally, lines 26-29 determine whether Justice Scalia voted for the petitioner or the respondent and save the result to the variable `y`. These four lines include the boolean logic necessary to make sense of the data we have collected thus far.

We are finally ready to begin extracting features of the data. We want to find features that we think will be helpful to the logistic regression. Since we want to make predictions about the future, these features should be information that would be known before the outcome of the case has been revealed. One feature that might prove useful is whether the lower court made a "liberal" or "conservative" decision. If the lower court made a "liberal" decision, then a conservative-leaning justice might be more likely to side with the petitioner and overturn the lower court's decision. Likewise, if the lower court made a "conservative" decision, then a conservative justice might be more likely to uphold the lower court's decision and side with the respondent. We acknowledge that determining whether a decision was "liberal" or "conservative" is subjective, but any data, however imperfect, might be useful to help our model learn.

```
31      # Feature: did the lower court make a liberal decision?
32      if row['lcDispositionDirection'] == '' or row['lcDispositionDirection']
            == '3':
33          continue
34      x1 = int(row['lcDispositionDirection'])
```

On line 32, we check whether the column is missing or is set to *unspecified* (`'3'`). If not, then the column is set to `'1'` if the decision was conservative and `'2'` if the decision was liberal. We convert this string into an integer and save it as our first feature in the variable `x1`.

At this point, we will cease to collect more features for the moment. Before doing so, we should check whether this feature alone was useful. We therefore complete the body of the loop by saving the feature (`x1`) and the outcome (`y`) to the lists `xs` and `ys` respectively.

```
36      # Save to the training data.
37      xs.append([x1])
38      ys.append(y)
```

We are now done with the body of the loop. The next lines of code can assume that the loop has fully assembled our training data into `xs` and `ys`, meaning we are ready to proceed with machine learning. As a final step, we will print the number of votes represented in our training data.

```
40 print('Votes in training data: {0}'.format(len(ys)))
```

When we run this program, the following output will print:

```
Votes in training data: 2739
```

Training a logistic regression model. Our training data is now stored in two lists, `xs` and `ys`. Each item in `xs` is a list of features corresponding to a particular vote; right now, these feature lists contain one item. Each item in `ys` is how Justice Scalia voted: `1` if for the petitioner and `0` if for the respondent.

Our first step when performing machine learning is to separate these lists into training data and test data. Recall from the first section of this chapter that we need to set aside some data that our model hasn't yet seen to evaluate how it performs.

```
42 # Shuffle the training data.
43 shuffles = zip(xs, ys)
44 random.shuffle( shuffles )
45 xs = []
46 ys = []
47 for shuffle in shuffles :
48     xs.append(shuffle[0])
49     ys.append(shuffle[1])
50
51 # Set aside 20% of the data as test data.
52 test = int(.8 * len(ys))
53 xs_train = xs[:test]
54 xs_test = xs[test:]
55 ys_train = ys[:test]
56 ys_test = ys[test:]
```

We want to set aside 20% of our dataset to be test data. Before we can do that, we need to put our dataset in random order. If we were to simply pick the test data to be the last 20% of the items in our dataset, then our test data would contain the 20% of votes that took place most recently. It is possible that the more recent votes look different than the older votes, meaning our test set has different behavior from our training set.

Lines 43 to 49 put the training data in random order. This is a surprisingly delicate process, since we want to make sure that each entry of `xs` is still paired with the corresponding entry of `ys`, even after randomizing the order of the two lists. To do so, we zip the lists into one list of tuples (line 43) and then shuffle

that list of tuples (line 44). Lines 45-49 then separate these tuples back into two distinct lists.

Lines 52 to 56 slice xs and ys into separate lists for testing and training, with 80% of the data for training and 20% of the data for testing. Line 51 determines the index at which that 80% break occurs, and lines 53 to 56 perform the slicing.

After all of that preparation, we are ready to train the model. Doing so takes only a couple of lines of code.

```
58 # Create the model.
59 model = sklearn.linear_model.LogisticRegression()
60
61 # Fit the model to the data.
62 model.fit(xs_train, ys_train)
63
64 # Determine how accurate the model was on the test data.
65 accuracy = model.score(xs_test, ys_test)
66 print('Accuracy: {0}'.format(accuracy))
```

Line 59 creates a `LogisticRegression` object. Line 62 trains that object by calling the `fit` method with the training inputs (the first argument) and the expected training outputs (the second argument). Finally, line 65 uses the `score` method to assess how accurately the model performed on the test inputs and outputs.

Notice that training and predicting with a logistic regression model took only three lines of code. The vast majority of our program involved reading data, extracting features, and putting the data in the correct form to be processed by the machine learning model. This experience is reflective of data analysis and machine learning in the real world: far more time goes into data preparation than to actual modeling and analysis.

Although we did not use this method in the code above, the `predict` method takes as its sole argument a list of features and outputs the model's prediction (`0` or `1`). For example, to see how the model predicts Scalia would vote had the lower court made a liberal decision, we could write:

```
model.predict([2])
```

(Recall that `2` is the Supreme Court Database's code for "liberal" for the data about how the lower court decided.

Returning to the task at hand, we can run the program and see the following output.

```
Votes in training data: 2739
Accuracy: 0.651459854015
```

In other words, the model correctly predicted Justice Scalia's vote about 65% of the time on the test data. Since we randomly select the training and test data, this number will likely vary between 60% and 70% depending on luck.

At first glance, this number is pretty good. Using only a single feature, we were able to predict Justice Scalia's vote two thirds of the time.

However, it turns out that there was an even simpler strategy that would have yielded equivalent results. In practice, justices tend to vote in favor of overturning lower courts more often than upholding lower courts. In other words, a simpler alternative model is that we always guess that Justice Scalia will vote for the petitioner. How accurately does this model perform?

```
68 # Count the number of votes for the petitioner in the test data.
69 for_petitioner = 0.0
70 for vote in ys_test:
71     if vote == 1:
72         for_petitioner += 1
73
74 simpler_accuracy = for_petitioner / len(ys_test)
75 print('Accuracy of simpler model: {0}'.format(simpler_accuracy))
```

Lines 69 to 72 count the number of votes for the petitioner in the test set. Lines 74 and 75 determine the overall fraction of the test set that involves votes for the petitioner and prints that number.

```
Votes in training data: 2739
Accuracy: 0.667883211679
Accuracy of simpler model: 0.583941605839
```

This data shows that our logistic regression model does achieve about an 8% gain over the simpler model, so we can confirm that we have made some progress.

Adding more features and iterating. It is always desirable to do even better. The Supreme Court Database offers one other feature that seems interesting: whether the majority ruled in a "liberal" or "conservative" way. We can use this information to determine whether the petitioner represented the "liberal" or "conservative" side. Since many justices tend to consistently vote for the "liberal" or "conservative" side, our model may be able to learn Justice Scalia's tendencies and better predict his vote.

We can build another feature. In the original Python file we have developed, the code below would be inserted at line 35.

```
# Feature: is the petitioner the liberal or conservative side?
if row['decisionDirection'] == '' or row['decisionDirection'] == '3':
    continue
liberal = row['decisionDirection'] == '2'

if petitioner and liberal: x2 = 1
elif petitioner and not liberal: x2 = 0
elif liberal and not petitioner: x2 = 0
else: x2 = 1
```

These nine lines create a new feature, `x2`, which has the value `1` if the petitioner was the liberal side and `0` if the petitioner was the conservative side. The first three lines of this code block throw out any rows where this column was either empty or unspecified (`'3'`). The fourth line creates a boolean variable that is `True` if the majority decision was liberal and `False` otherwise.

The last four lines of this code block perform the logic to determine whether the petitioner was the liberal or conservative side. Recall that the variable `petitioner` is `True` if the petitioner won or `False` otherwise.

Our last step is to update the former line 37 to include both features.

```
37      xs.append([x1, x2])
```

We can now run the model to see how well it performs.

```
Votes in training data: 2737
Accuracy: 0.647810218978
Accuracy of simpler model: 0.582116788321
```

Unfortunately, this additional feature didn't seem to provide any improvement over the sole feature we had before. Within the framework we have developed in this section, one could easily experiment with many other features, including the circuit or state at which the case was previously decided and the subject matter of the case. It is possible that some combination of these features will yield better results, or that a more powerful kind of model (like a neural network) is necessary to model justices more accurately.[1]

This process of iteratively trying new features or different classes of models is a key part of the machine learning workflow. It is very rare for the first attempt at choosing features or selecting a model class to succeed. Instead, building a model is an gradual process of exploration of which we performed two steps in this section.

18.3 Neural Networks

Considering the hype ascribed to neural networks, it may surprise you to learn that they are merely a small embellishment of the logistic regression models we discussed in the previous section. You now have all of the conceptual knowledge you need to understand neural networks. In this section, we will introduce the notion of a neural network and show you how to create and train a model that classifies handwritten digits. This learning task is a popular choice for introductory tutorials into neural network development.

[1]There is a rich body of research on predicting Supreme Court votes, much of which does indeed use more sophisticated models and data than we had available here. This FiveThirtyEight article summarizes the models and datasets people have applied to predict Supreme Court votes:
`https://fivethirtyeight.com/features/why-the-best-supreme-court-predictor-in-the-world-is-some-random-guy-in-queens/`

This chapter's standard caveat—that machine learning is a vast subject we can't possibly hope to cover in a single chapter—particularly applies to neural networks. Neural networks are especially unwieldy, and even experts often struggle to get them to learn effectively. Many experts will admit that training a neural network is more art than science. Furthermore, the state of the art is changing rapidly. Neural networks are still a topic of active research, so best practices are constantly evolving and this week's conventional wisdom is quite literally often out of date by next week.

With all of that said, neural networks are the foundation for all of today's most successful artificially intelligent systems, so it is vital to understand how they work and how to create them. This knowledge provides valuable insight into an increasingly pervasive technology that is often misrepresented or misunderstood in popular portrayals. Moreover, for those readers who are interested in delving into the vast, developing, and chaotic world of neural networks, this chapter provides a useful jumping off point.

For all of their flaws, neural networks are a demonstrably effective technique for taking on problems that, in prior decades, seemed far beyond the grasp of computer science. Not only are they here to stay, but they are likely to further grow in importance as we begin to fill in the myriad gaps in our scientific understanding of how they work. For that reason, we close out this chapter (and this book) by introducing the most exciting topic in modern machine learning: neural networks.

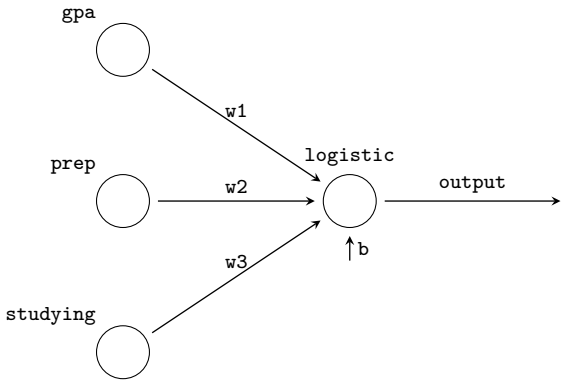
18.3.1 Neural Networks in Concept

From logistic regression to neural networks. The story of neural networks begins with logistic regression. Recall an example from our discussion of logistic regression: we want to predict whether a law student will pass the bar exam given three features: her law school GPA, whether she took a bar prep class, and how many hours she spent studying. Our logistic regression model for this learning task took the following form:

$$\text{logistic\_function}\left(\text{w1} * \text{gpa} + \text{w2} * \text{prep} + \text{w3} * \text{studying} + \text{b}\right)$$

This model takes three features: the student's law school GPA (`gpa`), whether the student took a bar prep class (`prep`), and the number of hours that the student spent studying (`studying`). The model also has three parameters, `w1`, `w2`, and `w3`, that control the extent to which each of these features is taken into account by the model. A fourth parameter, `b`, determines how much the logistic function will be shifted to the left or right. Training this model involves finding values for these four parameters that make accuracy as high as possible.

So far, all of this discussion is a review of Section 18.2. Below, we've rewritten the structure of the model as a flow chart.



The circles along the left from top to bottom are the three features from before: `gpa`, `prep`, and `studying`. The circle in the middle represents the logistic function. The arrow exiting from the right of the circle in the middle represents the output of the logistic function. This diagram is meant to represent the exact same logistic regression model as we just reintroduced. Let's see how it works.

The circle in the center representing the logistic function has four incoming arrows. One arrow goes from `gpa` to the logistic function and is labelled with `w1`. This arrow represents multiplying the `gpa` feature by its parameter `w1`, i.e., `gpa * w1`. Another arrow travels from `prep` to the logistic function and is labelled with `w2`, representing the value `prep * w2`. Similarly, the arrow from `studying` to the logistic function labelled with `w3` represents the value `studying * w3`. A fourth arrow labelled with `b` also points to the logistic function.

The four values that point to the logistic function are added together, producing the quantity

$$\text{gpa} * \text{w1} + \text{prep} * \text{w2} + \text{studying} * \text{w3} + \text{b}$$

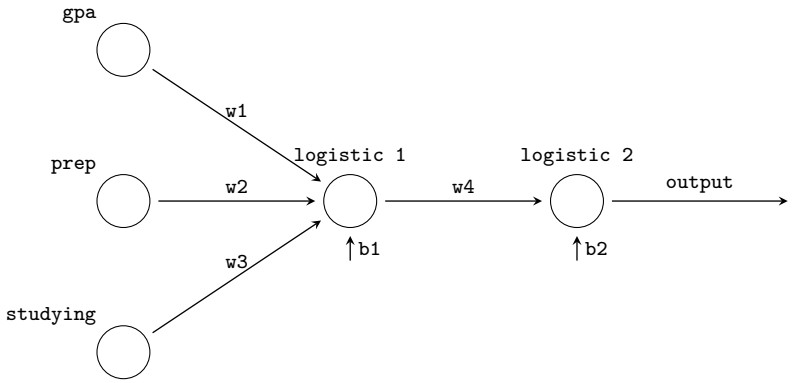
and then run through the logistic function, producing the output

$$\text{logistic\_function}\left(\text{w1} * \text{gpa} + \text{w2} * \text{prep} + \text{w3} * \text{studying} + \text{b}\right)$$

This output is represented by the arrow labelled `output` travelling rightward from the `logistic` circle.

This diagram is just the logistic regression equation rewritten in a different, visual form. It is identical to the formula we showed before. This diagram is useful because it shows the way data flows through the logistic regression. For example, this diagram shows how the `gpa` feature flows through the parameter `w1` before being combined with the other features and passed through the logistic function and how the output is derived from all of the features and parameters. Before moving on, study this diagram carefully to make sure you fully understand it. This diagram is your first neural network.

The key, counterintuitive insight behind neural networks is that it can be useful to perform logistic regression more than once. We can look at the output of logistic regression as just another feature. This new feature can, itself, be run through another logistic regression to produce another output. The diagram below shows this idea.



Just as before, our three features are the circles along the left. They are multipled by their corresponding `w`'s, added to the parameter `b1`, and fed into the logistic function (the circle labelled `logistic 1`). The result of this first logistic function, which we'll call `a1`, can be written as

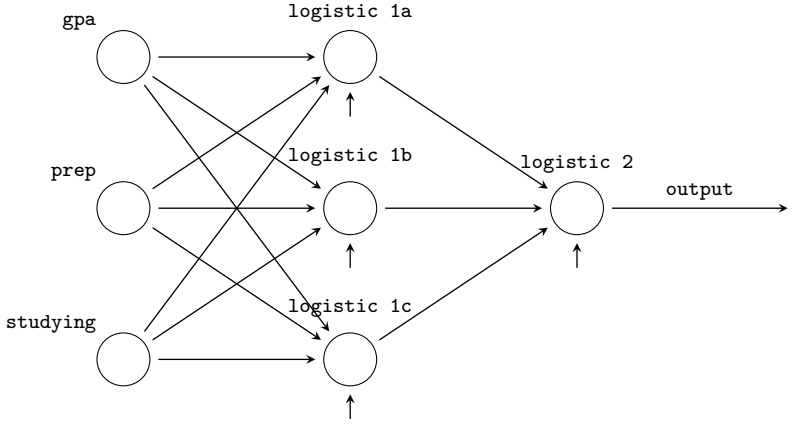
$$\text{a1} = \text{logistic\_function}(\text{gpa} * \text{w1} + \text{prep} * \text{w2} + \text{studying} * \text{w3} + \text{b1})$$

Before, `a1` was the output of our model. Here, however, the output of this first logistic function is the input to another logistic function. It is multiplied by another parameter, `w4`, added to the parameter `b2`, and run through another logistic function (the circle labelled `logistic 2`). The output of this second logistic function is the output of our model. In Python,

$$\text{a2} = \text{logistic\_function}(\text{a1} * \text{w4} + \text{b2})$$

This model now has six parameters: `w1`, `w2`, `w3`, and `b1` just as in the previous model, and `w4` and `b2` from the additional logistic function.

This is the key insight behind neural networks: we can compute many logistic regressions and wire them together to create more sophisticated models. The previous diagram shows that we can arrange these logistic regressions in sequence, one after the other. We can also compute multiple logistic regressions in parallel and feed them into another logistic regression:



This diagram has four logistic regressions. Three of the logistic regressions, the circles labelled `logistic 1a`, `logistic 1b`, and `logistic 1c`, are all computed in parallel. Each of the three features (`gpa`, `prep`, and `studying`) has an arrow to each of the the three logistic regressions, making nine arrows in total between the three features and the first three logistic regressions. Each of these arrows has its own separate `w` (which we have hidden to make the diagram readable).

In Python, the output of `logistic 1a`, which we will call `a1a`, is computed as follows:

a1a = logistic_function(gpa ∗ w1a + prep ∗ w2a + studying ∗ w3a + b1a)

This formula is a standard logistic regression with parameters `w1a`, `w2a`, `w3a`, and `b1a`.

The output of `logistic 1b`, which we call `a1b`, can be computed in similar fashion.

a1b = logistic_function(gpa ∗ w1b + prep ∗ w2b + studying ∗ w3b + b1b)

This is yet another standard logistic regression with its own parameters separate from the other logistic regressions.

The output of `logistic 1c` is similar.

a1c = logistic_function(gpa ∗ w1c + prep ∗ w2c + studying ∗ w3c + b1c)

The outputs of these logistic regressions, `a1a`, `a1b`, and `a1c`, are then fed into yet another logistic regression, `logistic 2`. This logistic function has its own set of parameters, `w1`, `w2`, `w3`, and `b`:

output = logistic_function(a1a ∗ w1 + a1b ∗ w2 + a1c ∗ w3 + b)

The result of this final logistic regression is the output of the overall model. This model is yet more complicated: it has four logistic regressions and 16 parameters.

Defining a neural network. A neural network computes logistic regression (or other similar functions) on combinations of features in an iterative fashion. These functions are arranged into a network (hence the name). The process of training involves searching for parameters that improve the overall accuracy of the model.

What advantage does a neural network have over logistic regression? Why is it any better to compute logistic regression four times rather than just once? In the previous example, we computed three seemingly identicial logistic regressions of the input features—why is this any better than doing it once? Through the process of training a neural network, each of these logistic regressions will start to specialize. In other words, since these logistic regressions are each just a small part of a much larger model, they don't have to solve the entire problem on their own. Instead, they can learn intermediate information that doesn't solve the problem entirely but serves as a step along the way to the solution. This intermediate information—the outputs of the first three logistic regressions—are

then fed into another logistic regression that can take advantage of this work to solve the remainder of the problem.

When the model is first created, all of the parameters are set to random values. This means that, even though the first three logistic regressions (`logistic 1a`, `logistic 1b`, and `logistic 1c`) are configured in the same way, they will initially produce completely different outputs. Initially, these outputs are likely to be nonsense. However, as the overall network learns, these logistic regressions will learn to output useful information as their parameters are slowly improved through the process of training. Since they were initially set to completely different, random values, they will learn to generate different kinds of intermediate information. The bottom line is that three logistic regressions will learn three different pieces of useful information, making the final logistic regression's job easier.

The networks we have considered so far are miniscule by modern standards. The network we will use to classify handwritten digits has 410 logistic regressions and nearly 270,000 parameters. Even this network is considerd tiny: industrial-strength networks for image classification or facial recognition might have tens of thousands of logistic regressions and billions of parameters. These gigantic models can take months to train. However, as you now know, the fundamental building block of all of these networks is logistic regression (or variations thereof).

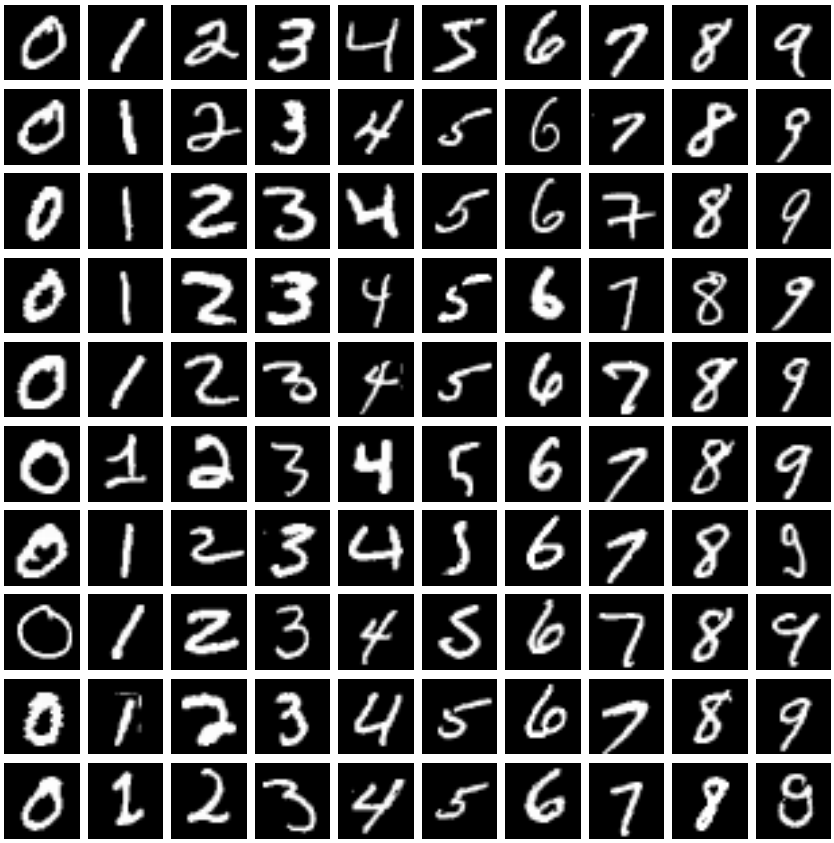
Terminology. The field of neural networks has developed its own lexicon for describing each of the components we have introduced so far. The features of the original data (like `gpa`, `prep`, and `studying`) are known as inputs. Each of the parameters that the features are multiplied by (the `w`'s) are known as weights. The additional `b` values are known as biases. The logistic regressions, which we have drawn as circles, are known as units, nodes, or neurons (we will use the term unit here). Although all of the units we have considered so far compute the logistic function, many neural networks replace the logistic function with other functions that seem to work better in certain contexts. The particular function that a unit computes is called its activation function.

The units of a neural network are typically arranged into layers. For example, the network we showed previously has three layers. The features (the circles for `gpa`, `prep`, and `studying` on the far left) are known as the input layer. The column of units that comes next (`logistic 1a`, `logistic 1b`, and `logistic 1c`) are the first layer. Every feature in the input layer is connected to every unit in the first layer. When two layers are connected in this fashion, they are said to be fully-connected. (In more exotic neural network configurations, the layers are not fully connected, however we will only consider fully-connected layers in this chapter.) The third column, which contains just the unit `logistic 2`, is known as the second layer. Since it is the last layer, it is also referred to as the output layer because its output is the output of the entire model.

Most neural networks have several layers of units between the input layer and the output layer. The layers between the input layer and the output layer are

known as hidden layers, as they connect to neither the output nor the input. Our handwritten digit classification network will have two hidden layers, one with 300 units and one with 100 units. Bigger networks for image classification might have between ten and twenty hidden layers, and researchers are experimenting with networks that have tens or hundreds of hidden layers. Networks with many layers are said to be deep; the term deep learning refers to performing machine learning with a deep neural network. There is no threshold for when a network goes from being "shallow" to being "deep," so the term "deep learning" has no practical significance beyond marketing.

Handwritten digit recognition. The National Institute of Standards and Technology (NIST) prepared a dataset containing thousands of pictures of handwritten digits. Some of the writers had excellent handwriting and the digits are easy to decipher. Others were less careful. Several early pioneers in neural networks research selected 70,000 pictures from this collection and compiled the MNIST dataset (`http://yann.lecun.com/exdb/mnist/`). Each of these pictures is a square (28 pixels by 28 pixels, making 784 pixels in total) containing a grayscale image of one of these digits. The machine learning problem is as follows: correctly identify as many of these digits as possible. Below are several examples images:



At first glance, this is a more challenging machine learning task than any we have encountered so far. However, we can apply the same methodology as we have in the past.

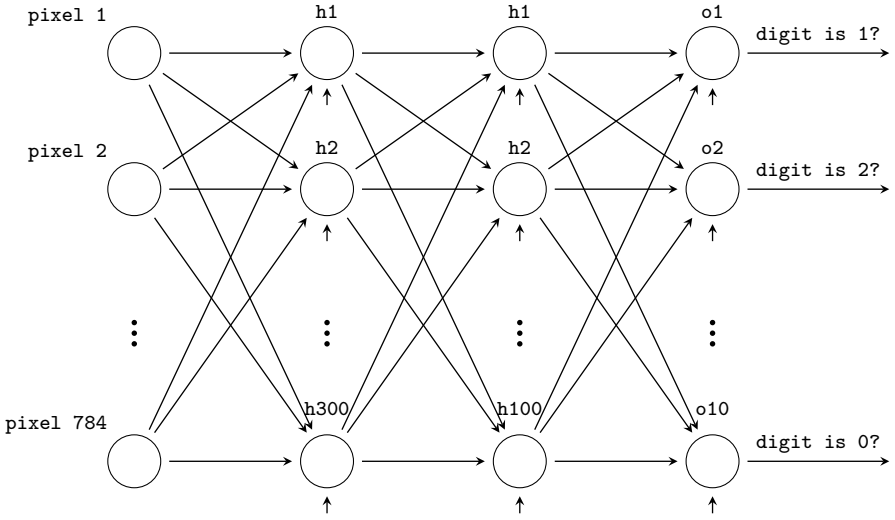
First, what is our task? We want to develop a model that can predict which digit (0 through 9) is contained within an image. This is a classification task. However, this classification problem is slightly harder than those we have considered in the past. All of the classification tasks we have discussed thus far have asked models to choose between two categories (will or won't pass the bar, will or won't be a profitable day for ice cream, will or won't vote for the petitioner). Here, the model must choose between ten categories: one for each digit that the image could contain.

Next, what are our features? Each picture in MNIST is made up of 784 pixels arranged in a 28 x 28 square. This pixel has a value between 0 and 1 representing its intensity from darkest (0) to lightest (1). Each of these pixels is one feature, meaning every image has 784 features.

Next, what is our training and test data? The MNIST dataset provides 70,000 images of digits. 60,000 of these images have been designated as training data and 10,000 have been designated as test data.

Now comes the hard part: what is our model family? We will use a neural

network, but one that is much bigger than anything we have considered thus far. The input layer of this network has 784 units—one for each feature of an image. The network has two hidden layers. The first hidden layer has 300 units and the second hidden layer has 100 units. The output layer of the network has ten units—one for each possible classification decision we can make. All of the layers are fully connected, meaning that each unit is connected to all of the units in the layer before it. Using the same diagrams as before, our network architecture is as follows:



The left column represents the input layer, which has 784 units containing the values of the 784 pixels in each image. Only three units are drawn; the elipses between `pixel 2` and `pixel 784` signify the remaining 781 pixels that did not fit on this page. Each of these input units is connected to every unit in the second column from the left, which represents the first hidden layer. These units are labeled `h1` through `h300`, with 297 of the units represented by ellipses. These units multiply their inputs by the corresponding weights, add a bias, and compute the logistic function. Since there are 784 units in the input layer and 300 units in the second hidden layer, there are 235,200 weights connecting the two layers.

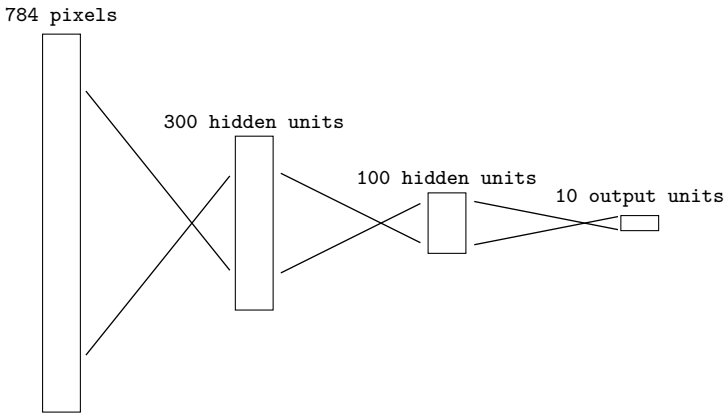
Each of the 300 units in the first hidden layer is connected to each of the 100 units in the second hidden layer (the third column from the left). Each unit in the second hidden layer multiplies the outputs of the first hidden layer by the corresponding weights, adds a bias, and computes the logistic function. Since there are 300 units in the first hidden layer and 100 units in the second hidden layer, there are 30,000 weights between the two.

Finally, each of the 100 units in the second hidden layer is connected to the ten units in the output layer (the last column on the right). There are 1000 weights between these two layers. The units in the output layer do not compute the logistic function. Instead, they compute a very similar function called the softmax function, which is like a ten-way logistic function. The softmax function

ensures that the values of the ten output units add up to 1. Each output unit can be thought of as outputting the network's confidence that the image contains a particular digit. For example, if the first output unit's value is `0.5`, the second output unit's value is `0.35`, and the fifth output unit's value is `0.15`, then the network thinks there is a 50% chance the digit is a 1, and 35% chance the digit is a 2, and a 15% chance the digit is a 5. In an ideal world, the network would be 100% sure that the digit contained the right number; that is, if the network is fed a picture of an 8, we would hope that the eighth output unit would have the value `1.0` and all the other output units would have the value `0.0`. In reality, this is rarely the case, and we say that the network's decision is the digit to which it gives the highest probability.

Reread the previous paragraph carefully—the concepts are subtle but very important.

Before we continue, here is one other way of summarizing the network.



Each vertical bar represents the units in a layer. The X patterns represent the fact that the layers are fully-connected to one another. The leftmost bar represents the 784 input units representing the 784 images in the input layer. The second bar from the left represents the 300 units in the first hidden layer. They are fully connected to the input layer. The third bar from the left represents the 100 units in the second hidden layer, which are fully connected to the first hidden layer. Finally, the bar on the right represents the ten output units, which are fully connected to the second hidden layer. The bars are roughly in proportion to the sizes of the layers, giving you a sense for the way the network distills the massive amount of information contained in the raw pixels down to the ten outputs.

Training a neural network. Now that we have a network architecture, we need to determine out how to train it. A neural network is trained by repeating the following step over and over again, tens of thousands of times, until the network performs accurately:

1. Show the network an image from the training set.

2. Examine the output that the network produced.

3. Compare the output that the network produced to the correct output. For example, the network might say it was 60% certain that the image contained a 1 and 40% certain that the image contained a 7. However, if the image contained a 9, then the correct output would have been to be 100% sure that the image conatined a 9.

4. Slightly tweak the network's parameters so that its output on this image would have been closer to the correct output.

By repeating this operation over and over again, the parameters will slowly find their way toward values that accurately classify handwritten digits. Each repetition of these steps is known as a training iteration.

In practice, updating the parameters after a training iteration is very slow.[2] Rather than show the network one image, we typically show it a small mini-batch of images (typically 50 to 100 images), average the network's accuracy across those images, and then update the parameters based on that average accuracy. Doing so helps the network find good parameters in fewer training iterations and ensures that one idiosyncratic image doesn't throw off all of the network's parameters.

In general, the neural net training process can be understood as optimization: we want to find the set of parameters that minimizes the number and extent of mistakes that the network makes. There are many different ways of measuring the severity of a network's mistakes, although exploring that detail is beyond the scope of this book. The particular way of measuring the severity of a mistake is referred to as a loss function. Neural network training aims to find parameters that "minimize the loss," meaning the network makes fewer, smaller mistakes.

Summary. You now understand the basics of building and training neural networks. A neural network is simply many logistic functions (or other, related functions) chained together to create a large network with many parameters. These functions, which are known as units, are organized into layers that collectively analyze the input features and, layer by layer, put together the information necessary to classify it. These networks can be exceedingly large: even the small network we introduced for handwritten digit recognition contains 270,000 parameters that need to be trained. Neural networks are trained by repeatedly showing them mini-batches of training examples and updating the parameters to improve the accuracy that the network would have obtained on those examples.

The information we have covered in this section is just the basics of neural networks. Neural networks are an exceedingly hot topic for research and engineering, meaning that new configurations, paradigms, training methods, and optimizations are constantly being introduced. This section has described the

[2] For the mathematically inclined, it involves computing the partial derivative of the difference between network's output and the correct output (specifically, the network's loss, which we will introduce momentarily) with respect to each of the network's 270,000 parameters individually.

basic foundations that will can serve as a jumping off point for further learning or as a body of knowledge for analyzing neural networks that appear in law and policy contexts. In the next section, we will conclude this chapter by building in Python the neural network we designed for handwritten digit recognition.

18.3.2 Neural Networks in Python (Keras)

In this section, we will build the handwriting-recognition neural network that we described in the previous section. To do so, we will use a library called `keras`, which makes creating and training small neural networks exceedingly easy. In fact, we will import the data and create, train, and evaluate the network in less than 30 lines of Python (not including comments and whitespace). As was our experience in Section 18.2, it will take as much work to import and clean the data as it will to build, train, and use the model.

Before we can begin to build neural networks, you will need to install three Python libraries:

1. `numpy`, a high-performance numerical processing library. If you followed Section 18.2, you have already installed `numpy`.

   $ *pip3 install numpy*

2. `tensorflow`, a low-level library for building and training neural networks developed by Google.

   $ *pip3 install tensorflow*

3. `keras`, a high-level library that makes `tensorflow` dramatically easier to use.

   $ *pip3 install keras*

Importing data. Just as we experienced in Section 18.2, it will take a certain amount of work to simply get the MNIST dataset into the right form for our neural network.

First, we need to import two libraries.

```
1 import numpy
2 import keras
```

The `numpy` library is necessary for some of the numerical processing that we will need to do. The `keras` library will build, train, and evaluate our neural network.

We can now access the MNIST data. Since this dataset is so common for training and evaluating neural networks, it comes built into `keras`.

```
4 # Import the MNIST training data.
5 mnist = keras.datasets.mnist.load_data()
```

Line 5 asks `keras` to download the dataset if it has not been downloaded previously and to load it into the variable `mnist`. The first time you run this statement, it may take a couple of minutes for MNIST to download. The variable `mnist` is a tuple with two elements. The first element is the training set and the second element is the test set. We separate those two elements on lines 6 and 7.

```
6  train = mnist[0]
7  test = mnist[1]
```

The training set contains the 60,000 images in the official MNIST training set, and the test set contains the 10,000 images in the official MNIST test set. Both of these are, themselves, tuples with two elements. The first element is the input features (that is, the images themselves) and the second element is the digit written in that picture (that is, the output we want the network to produce on that image). On lines 8 to 11, we separate out this data.

```
8  x_train = train[0]
9  y_train = train[1]
10 x_test = test[0]
11 y_test = test[1]
```

We finally have all of the data we need to train the network. Unfortunately, neither the x-values nor the y-values are in the proper form we need for our network. We will begin with the x-values, which have two problems. First of all, the x-values have the intensity of each pixel (how light or dark it is) stored as an integer between `0` and `255`. It will be much easier on the network if this value is between `0.0` and `1.0`. In short, we need to divide every pixel of every image by `255`.

Thankfully, the `numpy` library will make this job easier. The x-values and y-values are not stored as normal Python lists. Instead, they are stored as a special `numpy` type called an array. These `numpy` arrays are very similar to Python lists, but they have several features that make them easier to work with in numerical settings. In this case, we can divide the entire array by `255` in one step.

```
13 # Put the x-values in the proper form.
14 x_train = x_train / 255
15 x_test = x_test / 255
```

If we try to divide a list by `255`, Python will crash with an error message. If we divide a `numpy` array by `255`, `numpy` will understand our intent and divide every element in that array by `255`.

The second problem with our x-values is that they have the wrong shape. Right now, `x_train` is a `numpy` array with 60,000 elements—one for each image in the training set. Each of these elements is a 28 x 28 `numpy` array containing the pixels of the image arranged in a square. All in all, the shape of this array is said to be 60,000 x 28 x 28, since it has 60,000 images that are each 28 x 28. However, our network needs these pixels arranged in a single list of 784

elements. In other words, we want the shape of this array to be 60,000 x 784. Thankfully, `numpy` arrays make this transformation easy as well.

```
16 x_train = x_train.reshape([60000, 784])
17 x_test = x_test.reshape([60000, 784])
```

Each `numpy` array has a method called `reshape` that will return a new array containing a copy of the original array with its elements rearranged into the shape you want. It takes as its argument a list of the dimensions that you want the array to be converted to. The `reshape` method keeps all of the values of the array in the same order but changes the dimensions, so you have to use it with care to ensure that it does what you intend. In this case, however, it does exactly what we want.

The x-values are now in the proper form, but we still have to fix the y-values. The problem here is that each element of `y_train` and `y_test` is the integer that the corresponding picture contains. For example, if the picture had an image of the number eight, then the corresponding y-value would be `8`. However, as we discussed in the previous section, we want the y-value to contain the percentages that each of the network's output units should produce. In other words, the y-value we want in this case should be a list of ten elements with a `1` at index 8 and a `0` everwhere else: `[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]`. Here, `keras` will bail us out. It has a function designed for exactly these circumstances.

```
19 # Put the y-values into the proper form.
20 y_train = keras.utils.np_utils.to_categorical(y_train)
21 y_test = keras.utils.np_utils.to_categorical(y_test)
```

The `keras.utils.np_utils.to_categorical` converts every element of `y_test` and `y_train` into the correct form. We are finally ready to train the model.

Building a neural network model. The `keras` library makes building a neural network very easy. First, we create a `keras.models.Sequential` object representing an initially empty network. We can then build the network layer by layer.

```
23 # Build the model.
24 model = keras.models.Sequential()
```

We can then add the first fully-connected hidden layer, which has 300 hidden units. The `keras` library refers to a fully-connected layer as a "dense" layer (since it is densely-connected to the previous layer.).

```
25 model.add(keras.layers.Dense(300, activation='sigmoid', input_shape=[784]))
```

Line 25 creates an object that describes the layer we wish to add to the network (a `keras.layers.Dense` object) and then adds it to the network by passing it as an argument to the `add` method of `model`. The first argument of the `keras.layers.Dense` function is the number of hidden units in the layer. As we discussed in the previous section, the first hidden layer has 300 units.

We also provide two keyword arguments: `activation` and `input_shape`. The `activation` argument specifies which activation function we want to use for this layer. Recall that an activation function is the function that the unit runs all of its inputs through after adding them together. In the model we discussed in the previous section, we always used the logistic function as our activation function. Here, we do the same thing. The sigmoid function is just another name for the logistic function; the term sigmoid is more popular in the neural network community.

The other keyword argument, `input_shape`, specifies the shape of each input that will be provided to this layer from the previous layer. Since the first hidden layer immediately follows the input layer, it should expect to see inputs with 784 elements. This argument has to be provided as a one-element list, since some kinds of networks work with two-dimensional inputs (like images). The `input_shape` argument only needs to be passed to the first hidden layer, since it has no other way of knowing how big the input is. Subsequent layers can figure this out by looking at previous layers.

We repeat the same exercise for the next hidden layer.

```
26 model.add(keras.layers.Dense(100, activation='sigmoid'))
```

The second hidden layer has 100 hidden units, and we change the argument accordingly. It still has a sigmoid activation function. This time, we do not need to provide an `input_shape` argument.

Finally, we create the output layer.

```
27 model.add(keras.layers.Dense(10, activation='softmax'))
```

This layer has ten units—one for each of the possible digits. As you'll recall from the previous section, this layer needs a special softmax activation function that ensures that the values of the output units always adds up to 1.

In four lines of Python, our network is now complete. The last step is to compile the network. The `keras` library is just a convenient way of writing networks that will be created and run in the lower-level `tensorflow` library. In other words, `keras` needs to compile the network we just created into a form acceptable to `tensorflow`.

```
28 model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['
     accuracy'])
```

To compile the model, we use the `model` object's `compile` method. We need to pass this method several arguments that are necessary for the model to train correctly but that will nearly always be the same whenever you use `keras`. The `loss` argument specifies the loss function that the training process will use to measure whether the network has improved. In classification tasks, it is common to use a loss function known as cross entropy loss to which the argument `'categorical_crossentropy'` refers. We do not go into the details of cross entropy loss in this book, but it is a good choice in most classification settings.

The `optimizer` argument specifies how the training process should modify the network's parameters during each training iteration. The adam algorithm is a common choice, hence the argument `'adam'`.

Finally, the `metrics` argument specifies any additional information we want `keras` to tell us about the network as it trains. In our case, the most important metric is accuracy—how often did the network make the right choice.

Training the neural network model. We are now ready to train the neural network. Just as with `scikit-learn`, training takes only one line of code.

```
30 model.fit(x_train, y_train, batch_size=100, epochs=10, verbose=1)
```

The `fit` method tells `keras` to train the model. The first two arguments to the `fit` method are the training set's inputs and outputs. We also need to tell `keras` a few other pieces of information. The `batch_size` argument tells `keras` how many training images to consider in each training iteration; 100 is a reasonable choice.

The `epochs` argument tells `keras` how long to train the network. As `keras` trains the network, it will iterate over the entire training set until it runs out of images. Each pass through the training set is known as a epoch. The `epochs` argument tells `keras` how many times to iterate over the training set before training is complete; 10 epochs is a reasonable choice for this network and dataset.

Notice that, for both of the previous two arguments, we have simply told you what reasonable values were without giving you an explanation for why. The same is true for many of the numbers in this section: why does the network have two hidden layers of 300 and 100 units each? Why not more or less? Designing and training neural networks remains more art than science. Over the years, neural network engineers have found these numbers to be effective for training a model for MNIST. For other learning tasks on other datasets, different numbers have been found to work well. When you take on a new machine learning problem, you will have to find the right numbers for yourself. In many ways, neural networks are still in their infancy, and we have yet to develop a general set of principles for guiding neural network development beyond simple trial and error.

Returning to the task at hand, the last argument of the `fit` method, `verbose`, tells `keras` that we want it to give us detailed diagnostic information as the network trains.

When you call this method, the network will begin to train. Training a neural network is a complicated endeavor, and it will take anywhere from two to five minutes for your network to train. Since we used the `verbose` argument, `keras` will keep us posted on how training is progressing.

```
60000/60000 [==============================] − 5s 88us
    /step − loss: 0.5551 − acc: 0.8583
Epoch 2/10
```

```
60000/60000 [==============================] - 5s 76us
    /step - loss: 0.2184 - acc: 0.9361
Epoch 3/10
60000/60000 [==============================] - 4s 73us
    /step - loss: 0.1593 - acc: 0.9529
Epoch 4/10
43600/60000 [====================>.........] - ETA: 1s - loss:
    0.1263 - acc: 0.9629
```

The output during training should look something like the above. Each line is a progress bar showing how many training examples `keras` has used during the curent epoch. At the end of each epoch, it outputs two values, the `loss` and `acc` (i.e., accuracy) as computed on the training set. With each epoch, the loss should go down as the network makes fewer mistakes. Correspondingly, the accuracy should go up as the network makes fewer mistakes. By the end of the first epoch, the network has already reached 85% accuracy on the training set. By the end of the fourth epoch, it has reached 96%.

While accuracy on the training set is important, it does not tell us anything about how the network will perform on new inputs. After all, if the network simply memorized the training set, it could get perfect accuracy on the training set while being unable to make any predictions about new inputs. This is where the test set comes in. The last step in the machine learning process is to evaluate our model on inputs it has never seen before—the test set. The `model` object has an `evaluate` method for doing just that.

```
32 # Evaluate the model on the test set.
33 results = model.evaluate(x_test, y_test)
34 print('Test loss: {0}'.format(results[0]))
35 print('Test accuracy: {0}'.format(results[1]))
```

The `evaluate` method takes two arguments: the test inputs (`x_test`) and the expected outputs (`y_test`). It returns a tuple with two elements. The first element is the loss on the test set, and the second element is the accuracy on the test set. Lines 34 and 35 print both values. Below is the output that `keras` produces after training and testing.

```
60000/60000 [==============================] - 5s 89us
    /step - loss: 0.5815 - acc: 0.8506
Epoch 2/10
60000/60000 [==============================] - 5s 78us
    /step - loss: 0.2187 - acc: 0.9366
Epoch 3/10
60000/60000 [==============================] - 5s 86us
    /step - loss: 0.1609 - acc: 0.9525
Epoch 4/10
60000/60000 [==============================] - 5s 89us
    /step - loss: 0.1231 - acc: 0.9632
Epoch 5/10
```

```
60000/60000 [==============================] - 5s 91us
    /step - loss: 0.0958 - acc: 0.9726
Epoch 6/10
60000/60000 [==============================] - 5s 90us
    /step - loss: 0.0768 - acc: 0.9776
Epoch 7/10
60000/60000 [==============================] - 5s 90us
    /step - loss: 0.0612 - acc: 0.9818
Epoch 8/10
60000/60000 [==============================] - 5s 89us
    /step - loss: 0.0501 - acc: 0.9856
Epoch 9/10
60000/60000 [==============================] - 5s 85us
    /step - loss: 0.0402 - acc: 0.9885
Epoch 10/10
60000/60000 [==============================] - 5s 84us
    /step - loss: 0.0328 - acc: 0.9910
10000/10000 [==============================] - 1s 61us
    /step
Test loss: 0.06755954547743313
Test accuracy: 0.9786
```

At the end of training, the accuracy on the test set is 97.86%, which is excellent for a seemingly tricky learning task. However, notice that at the end of the last epoch, the accuracy on the training set is 99.10%. Why is the accuracy on the training set measurably higher than on the test set? By the end of the tenth epoch, the model has seen every example in the training set. It has learned a model that is exceedingly good at classifying those examples. However, it has gotten so used to the training set that it is unable to replicate that performance on new examples. This is a phenomenon called overfitting, where the model learns the nuances of the training set so closely that it becomes slightly worse at classifying new examples.

Before we close this section, there is one final method that you will find useful. If you have new data and want to see the predictions that your model makes, you can use the `predict` method. It takes as its sole argument an x-value (in this case, an image), and produces its predicted y-value (in this case a list of ten elements containing the model's certainty about whether the image is of each digit).

```
model.predict(x)
```

With that, you now know all of the basics of creating, training, and testing neural networks with the `keras` library. There is a far larger world of neural network designs and techniques that you can explore, and you now have a solid jumping-off point from which to begin.

18.3.3 Challenges in Neural Networks

Neural networks date back to the dawn of modern computing. The initial concept that evolved into modern neural networks, known as the perceptron, was invented in the 1950s. Neural network development has proceeded in bursts since then, with the most recent resurgence, which began in 2011, leading to their current prominence.

In spite of these decades of study, we still have a poor understanding of the kinds of neural networks that are commonly in use today. In this section, we will explore why this is the case and how this manifests in several different categories of practical challenges.

In general, all of the challenges with modern neural nets trace back to their sheer scale. The distinguishing quality of the current generation of neural networks is that they involve exceedingly large models (millions of parameters) and trained on exceedingly large datasets (hundreds of millions of training examples). We currently do not understand how to translate such complicated models into high-level algorithms that make sense to humans.

As an analogy, consider asking Larry the lawyer for his favorite flavor of ice cream. He tells you that his favorite flavor is vanilla. Analogously, you can supply a picture to a neural net and it will tell you what object it contains.

Now suppose you wish to understand *why* Larry's favorite flavor is vanilla. Rather than simply tell you, Larry instead hands you a scan of his brain showing the connections between the tens of billions of its constituent neurons. Then he says, "Here is a map of my brain, the biological algorithm that decided I that I like vanilla. You figure out why it made that decision." At the moment, understanding a neural network's decisionmaking process is as difficult as understanding Larry's preference for ice cream by looking at the internal wiring of his brain.

As neural network-based artificially intelligent systems continue to proliferate into contexts where they can have immensely consequential impacts on people's lives, this situation poses a variety of concerning practical challenges. With that said, it is important to remember that this is simply the current state of research, not a permanent condition of using neural networks. We have been using neural networks for large scale machine learning applications for less than a decade, so in many ways this generation of the technology remains in its infancy. It is common for powerful new technologies to be applied long before they are fully understood, and neural networks are no exception. Although many of the challenges we discuss in this section may seem disquieting or even dire, the state of the art is developing rapidly and it is entirely possible that we may solve some or all of these problems in the near future.

Explainability and interpretability. Neural networks are increasingly tasked with making consequential decisions, for example, who gets hired and who gets credit. In many of these settings, we wish to know why a neural network made a particular decision. When a self-driving car inexplicably fails to apply the brakes at a red light, it is important to be able to understand why it did so.

As we have already insinuated, the decisions made by neural networks currently defy explanation. We do not know how to translate complicated connections between thousands of units into human-comprehensible insights about the network's decisionmaking process.

It is possible that a neural network has learned a simple algorithm that we simply do not know how to extract. It is also possible that a neural network has learned an algorithm that is very effective at solving the problem at hand but would seem counterintuitive to humans or, worse, is too sophisticated for a human to easily grasp. Finally, it is possible that the network has simply memorized the training set or has learned some heuristic that works on the training and test sets but doesn't work in general. Right now, we have no way of telling the difference between these three scenarios.

To add one last complication to the question of explaining a neural network's decisionmaking process, we have yet to clearly articulate what it means—in technical terms—to produce a "human-comprehensible explanation." Different kinds of explanations make sense in different contexts to different humans, and different circumstances may demand vastly different levels of detail. For example, consider one possible form of explanation for a neural network's decision: *The network had two hidden layers of 300 and 100 hidden units with sigmoid activation functions. It was trained on 60,000 images of handwritten digits for 10 epochs. Once the parameters were so trained, we fed it an image and it decided it contained the number three.* This certain explains why a network made a decision, but not in a fashion that is useful.

In summary, the sheer complexity of a neural network makes it difficult to understand why it made a particular decision. Right now, we have yet to even define what a "good" explanation would look like.

Debug-ability. Suppose you discover a flaw in a neural network. For example, whenever a self-driving car encounters a particularly tricky driving situation, it makes the wrong decision. You want to somehow "correct" this behavior in the neural network.

Right now, we are unable to reconfigure or "rewire" the neural network to perform these sorts of fixes, owing to the fact that we do not know how to understand the high-level algorithm (if any) reflected in the neural network's underlying circuitry. In other words, even if we find a bug in a neural network, we have no way to fix the network to remove this bug.

The only way to attempt to fix a network is to train it more (or to retrain it from scratch). The training set should be updated to include examples that show the network the correct decisions to make in circumstances where it is currently making mistakes. However, there is no guarantee that this corrective training will necessarily help the network avoid these mistakes. We do not fully understand the impact that training has on the network; it is possible that the network will ignore these new training examples or that the lessons that the network learns from these examples will come at the cost of forgetting other knowledge.

In summary, even when we find bugs in neural networks, we have no reliable way to fix them.

Reliability and security. Neural networks can fail in unexpected, catastrophic ways. For example, researchers have shown that neural networks can easily be fooled. Supose you have a picture that both you and the network agree contains a cat. This picture can be slightly modified in a carefully-calibrated way to create a new picture that looks unmodified to you but looks like a dog to the network.

In general, since we do not understand how to interpret the neural network's circuitry, it is possible that the network has learned bad behaviors that go completely undetected until they are triggered by the wrong input. In other words, a perfect storm of unlucky circumstances may trigger some catastrophic behavior that was previously invisible.

Worse, an attacker can exploit this fact to design bad behaviors and secretly embed them into the network. For example, an attacker can "poison" the network's training data: he can train the network underlying a self-driving car to suddenly accelerate every time it sees a stop sign with a special sticker attached to it. When the car sees a normal stop sign, it will behave as expected and this hidden behavior will go undetected. However, when the attacker puts one of these special stickers on a stop sign, the car will suddenly accelerate.

Bias. Neural networks (and all machine learning systems) aim to emulate their training sets. However, most training sets come from real-world data, and this data inherits the implicit biases surrounding its creation. Networks that learn from this biased data will, in turn, be biased. Right now, researchers are working to determine what it means for a network to be biased, how we can measure the extent of a network's bias, and how we can train networks in ways that eliminate bias.

18.4 Summary

This chapter attempted to distill an enormous topic down to several dozen pages. In this chapter, you learned the definition of machine learning, the high-level process that it follows, and two concrete techniques for performing machine learning. Logistic regression is a relatively simple method that can learn to separate inputs into two different categories. By connecting many logistic regressions together, you can build a neural network, a dramatically more powerful (and more complicated and unwieldy) technique for building models. This chapter gave you both a conceptual introduction to all of these topics and concrete Python knowledge that can serve as a foundation for further exploration.

With the end of this chapter, your journey through this text comes to a close. Over the preceding pages, you have investigated what it means to program, written your first Python programs, learned the core of the Python language,

and studied several ways you can make use of the ability to program in your day-to-day life as a lawyer. This book is only the start of your journey as a programmer, and there are numerous paths you can take to sharpen your skills as a software engineer, learn new languages (e.g., Java or C++), target new platforms (e.g., websites or smartphone apps), find new applications, or apply your technical knowledge to problems in the legal and policy spheres.

We are honored that you began your journey with us, and we wish you the best in your future programming endeavors.