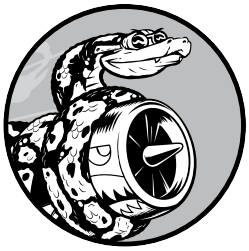
1

GETTING STARTED



In this chapter, you'll run your first Python program, *hello_world.py*. First, you'll need to check whether a recent version of Python is installed on your computer; if it isn't, you'll install it. You'll also install a text editor to work with your Python programs. Text editors recognize Python code and highlight sections as you write, making it easy to understand your code's structure.

Setting Up Your Programming Environment

Python differs slightly on different operating systems, so you'll need to keep a few considerations in mind. In the following sections, we'll make sure Python is set up correctly on your system.

Python Versions

Every programming language evolves as new ideas and technologies emerge, and the developers of Python have continually made the language more versatile and powerful. As of this writing, the latest version is Python 3.11, but everything in this book should run on Python 3.9 or later. In this section, we'll find out if Python is already installed on your system and whether you need to install a newer version. Appendix A contains additional details about installing the latest version of Python on each major operating system as well.

Running Snippets of Python Code

You can run Python's interpreter in a terminal window, allowing you to try bits of Python code without having to save and run an entire program.

Throughout this book, you'll see code snippets that look like this:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
```

The three angle brackets (>>>) prompt, which we'll refer to as a *Python prompt*, indicates that you should be using the terminal window. The bold text is the code you should type in and then execute by pressing ENTER. Most of the examples in this book are small, self-contained programs that you'll run from your text editor rather than the terminal, because you'll write most of your code in the text editor. But sometimes, basic concepts will be shown in a series of snippets run through a Python terminal session to demonstrate particular concepts more efficiently. When you see three angle brackets in a code listing, you're looking at code and output from a terminal session. We'll try coding in the interpreter on your system in a moment.

We'll also use a text editor to create a simple program called *Hello World!* that has become a staple of learning to program. There's a long-held tradition in the programming world that printing the message `Hello world!` to the screen as your first program in a new language will bring you good luck. Such a simple program serves a very real purpose. If it runs correctly on your system, then any Python program you write should work as well.

About the VS Code Editor

VS Code is a powerful, professional-quality text editor that's free and beginner-friendly. VS Code is great for both simple and complex projects, so if you become comfortable using it while learning Python, you can continue using it as you progress to larger and more complicated projects. VS Code can be installed on all modern operating systems, and it supports most programming languages, including Python.

Appendix B provides information on other text editors. If you're curious about the other options, you might want to skim that appendix at this

point. If you want to begin programming quickly, you can use VS Code to start. Then you can consider other editors, once you've gained some experience as a programmer. In this chapter, I'll walk you through installing VS Code on your operating system.

NOTE *If you already have a text editor installed and you know how to configure it to run Python programs, you are welcome to use that editor instead.*

Python on Different Operating Systems

Python is a cross-platform programming language, which means it runs on all the major operating systems. Any Python program you write should run on any modern computer that has Python installed. However, the methods for setting up Python on different operating systems vary slightly.

In this section, you'll learn how to set up Python on your system. You'll first check whether a recent version of Python is installed on your system, and install it if it's not. Then you'll install VS Code. These are the only two steps that are different for each operating system.

In the sections that follow, you'll run *hello_world.py* and troubleshoot anything that doesn't work. I'll walk you through this process for each operating system, so you'll have a Python programming environment that you can rely on.

Python on Windows

Windows doesn't usually come with Python, so you'll probably need to install it and then install VS Code.

Installing Python

First, check whether Python is installed on your system. Open a command window by entering `command` into the Start menu and clicking the **Command Prompt** app. In the terminal window, enter `python` in lowercase. If you get a Python prompt (>>>) in response, Python is installed on your system. If you see an error message telling you that `python` is not a recognized command, or if the Microsoft store opens, Python isn't installed. Close the Microsoft store if it opens; it's better to download an official installer than to use Microsoft's version.

If Python is not installed on your system, or if you see a version earlier than Python 3.9, you need to download a Python installer for Windows. Go to *https://python.org* and hover over the **Downloads** link. You should see a button for downloading the latest version of Python. Click the button, which should automatically start downloading the correct installer for your system. After you've downloaded the file, run the installer. Make sure you select the option **Add Python to PATH**, which will make it easier to configure your system correctly. Figure 1-1 shows this option selected.

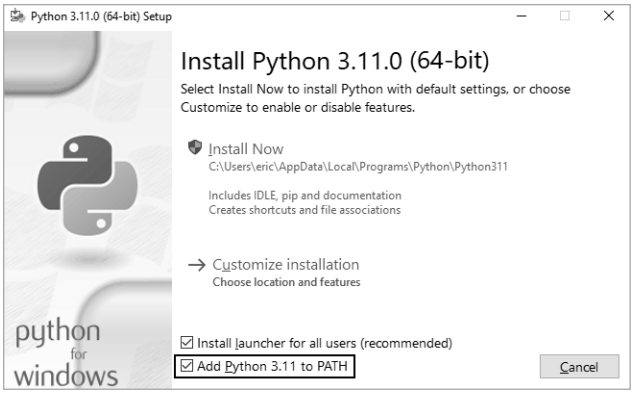


Figure 1-1: Make sure you select the checkbox labeled Add Python to PATH.

Running Python in a Terminal Session

Open a new command window and enter **`python`** in lowercase. You should see a Python prompt (>>>), which means Windows has found the version of Python you just installed.

```
C:\> python
Python 3.x.x (main, Jun . . . , 13:29:14) [MSC v.1932 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOTE *If you don't see this output or something similar, see the more detailed setup instructions in Appendix A.*

Enter the following line in your Python session:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

You should see the output `Hello Python interpreter!` Anytime you want to run a snippet of Python code, open a command window and start a Python terminal session. To close the terminal session, press CTRL-Z and then press ENTER, or enter the command **`exit()`**.

Installing VS Code

You can download an installer for VS Code at *https://code.visualstudio.com*. Click the **Download for Windows** button and run the installer. Skip the following sections about macOS and Linux, and follow the steps in "Running a Hello World Program" on page 9.

Python on macOS

Python is not installed by default on the latest versions of macOS, so you'll need to install it if you haven't already done so. In this section, you'll install the latest version of Python, and then install VS Code and make sure it's configured correctly.

NOTE *Python 2 was included on older versions of macOS, but it's an outdated version that you shouldn't use.*

Checking Whether Python 3 Is Installed

Open a terminal window by going to **Applications ▸ Utilities ▸ Terminal**. You can also press ⌘-spacebar, type **`terminal`**, and then press ENTER. To see if you have a recent enough version of Python installed, enter **`python3`**. You'll most likely see a message about installing the *command line developer tools*. It's better to install these tools after installing Python, so if this message appears, cancel the pop-up window.

If the output shows you have Python 3.9 or a later version installed, you can skip the next section and go to "Running Python in a Terminal Session." If you see any version earlier than Python 3.9, follow the instructions in the next section to install the latest version.

Note that on macOS, whenever you see the `python` command in this book, you need to use the `python3` command instead to make sure you're using Python 3. On most macOS systems, the `python` command either points to an outdated version of Python that should only be used by internal system tools, or it points to nothing and generates an error message.

Installing the Latest Version of Python

You can find a Python installer for your system at *https://python.org*. Hover over the **Download** link, and you should see a button for downloading the latest version of Python. Click the button, which should automatically start downloading the correct installer for your system. After the file downloads, run the installer.

After the installer runs, a Finder window should appear. Double-click the *Install Certificates.command* file. Running this file will allow you to more easily install additional libraries that you'll need for real-world projects, including the projects in the second half of this book.

Running Python in a Terminal Session

You can now try running snippets of Python code by opening a new terminal window and typing **`python3`**:

```
$ python3
Python 3.x.x (v3.11.0:eb0004c271, Jun . . . , 10:03:01)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This command starts a Python terminal session. You should see a Python prompt (>>>), which means macOS has found the version of Python you just installed.

Enter the following line in the terminal session:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

You should see the message `Hello Python interpreter!`, which should print directly in the current terminal window. You can close the Python interpreter by pressing CTRL-D or by entering the command **`exit()`**.

NOTE *On newer macOS systems, you'll see a percent sign (%) as a terminal prompt instead of a dollar sign ($).*

Installing VS Code

To install the VS Code editor, you need to download the installer at *https://code.visualstudio.com*. Click the **Download** button, and then open a **Finder** window and go to the **Downloads** folder. Drag the **Visual Studio Code** installer to your Applications folder, then double-click the installer to run it.

Skip over the following section about Python on Linux, and follow the steps in "Running a Hello World Program" on page 9.

Python on Linux

Linux systems are designed for programming, so Python is already installed on most Linux computers. The people who write and maintain Linux expect you to do your own programming at some point, and encourage you to do so. For this reason, there's very little to install and only a few settings to change to start programming.

Checking Your Version of Python

Open a terminal window by running the Terminal application on your system (in Ubuntu, you can press CTRL-ALT-T). To find out which version of Python is installed, enter **`python3`** with a lowercase *p*. When Python is installed, this command starts the Python interpreter. You should see output indicating which version of Python is installed. You should also see a Python prompt (>>>) where you can start entering Python commands:

```
$ python3
Python 3.10.4 (main, Apr  . . . , 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This output indicates that Python 3.10.4 is currently the default version of Python installed on this computer. When you've seen this output, press CTRL-D or enter **`exit()`** to leave the Python prompt and return to a

terminal prompt. Whenever you see the `python` command in this book, enter `python3` instead.

You'll need Python 3.9 or later to run the code in this book. If the Python version installed on your system is earlier than Python 3.9, or if you want to update to the latest version currently available, refer to the instructions in Appendix A.

Running Python in a Terminal Session

You can try running snippets of Python code by opening a terminal and entering `python3`, as you did when checking your version. Do this again, and when you have Python running, enter the following line in the terminal session:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

The message should print directly in the current terminal window. Remember that you can close the Python interpreter by pressing CTRL-D or by entering the command `exit()`.

Installing VS Code

On Ubuntu Linux, you can install VS Code from the Ubuntu Software Center. Click the Ubuntu Software icon in your menu and search for *vscode*. Click the app called **Visual Studio Code** (sometimes called *code*), and then click **Install**. Once it's installed, search your system for *VS Code* and launch the app.

Running a Hello World Program

With a recent version of Python and VS Code installed, you're almost ready to run your first Python program written in a text editor. But before doing so, you need to install the Python extension for VS Code.

Installing the Python Extension for VS Code

VS Code works with many different programming languages; to get the most out of it as a Python programmer, you'll need to install the Python extension. This extension adds support for writing, editing, and running Python programs.

To install the Python extension, click the Manage icon, which looks like a gear in the lower-left corner of the VS Code app. In the menu that appears, click **Extensions**. Enter `python` in the search box and click the **Python** extension. (If you see more than one extension named *Python*, choose the one supplied by Microsoft.) Click **Install** and install any additional tools that your system needs to complete the installation. If you see a message that you need to install Python, and you've already done so, you can ignore this message.

NOTE *If you're using macOS and a pop-up asks you to install the* command line developer tools*, click* ***Install****. You may see a message that it will take an excessively long time to install, but it should only take about 10 or 20 minutes on a reasonable internet connection.*

Running hello_world.py

Before you write your first program, make a folder called *python_work* on your desktop for your projects. It's best to use lowercase letters and underscores for spaces in file and folder names, because Python uses these naming conventions. You can make this folder somewhere other than the desktop, but it will be easier to follow some later steps if you save the *python_work* folder directly on your desktop.

Open VS Code, and close the **Get Started** tab if it's still open. Make a new file by clicking **File ▸ New File** or pressing CTRL-N (⌘-N on macOS). Save the file as *hello_world.py* in your *python_work* folder. The extension *.py* tells VS Code that your file is written in Python, and tells it how to run the program and highlight the text in a helpful way.

After you've saved your file, enter the following line in the editor:

hello_world.py

```
print("Hello Python world!")
```

To run your program, select **Run ▸ Run Without Debugging** or press CTRL-F5. A terminal screen should appear at the bottom of the VS Code window, showing your program's output:

```
Hello Python world!
```

You'll likely see some additional output showing the Python interpreter that was used to run your program. If you want to simplify the information that's displayed so you only see your program's output, see Appendix B. You can also find helpful suggestions about how to use VS Code more efficiently in Appendix B.

If you don't see this output, something might have gone wrong in the program. Check every character on the line you entered. Did you accidentally capitalize `print`? Did you forget one or both of the quotation marks or parentheses? Programming languages expect very specific syntax, and if you don't provide that, you'll get errors. If you can't get the program to run, see the suggestions in the next section.

Troubleshooting

If you can't get *hello_world.py* to run, here are a few remedies you can try that are also good general solutions for any programming problem:

- When a program contains a significant error, Python displays a *traceback*, which is an error report. Python looks through the file and tries to identify the problem. Check the traceback; it might give you a clue as to what issue is preventing the program from running.

- Step away from your computer, take a short break, and then try again. Remember that syntax is very important in programming, so something as simple as mismatched quotation marks or mismatched parentheses can prevent a program from running properly. Reread the relevant parts of this chapter, look over your code, and try to find the mistake.
- Start over again. You probably don't need to uninstall any software, but it might make sense to delete your *hello_world.py* file and re-create it from scratch.
- Ask someone else to follow the steps in this chapter, on your computer or a different one, and watch what they do carefully. You might have missed one small step that someone else happens to catch.
- See the additional installation instructions in Appendix A; some of the details included in the Appendix may help you solve your issue.
- Find someone who knows Python and ask them to help you get set up. If you ask around, you might find that you unexpectedly know someone who uses Python.
- The setup instructions in this chapter are also available through this book's companion website at *https://ehmatthes.github.io/pcc_3e*. The online version of these instructions might work better because you can simply cut and paste code and click links to the resources you need.
- Ask for help online. Appendix C provides a number of resources, such as forums and live chat sites, where you can ask for solutions from people who've already worked through the issue you're currently facing.

Never worry that you're bothering experienced programmers. Every programmer has been stuck at some point, and most programmers are happy to help you set up your system correctly. As long as you can state clearly what you're trying to do, what you've already tried, and the results you're getting, there's a good chance someone will be able to help you. As mentioned in the introduction, the Python community is very friendly and welcoming to beginners.

Python should run well on any modern computer. Early setup issues can be frustrating, but they're well worth sorting out. Once you get *hello_world.py* running, you can start to learn Python, and your programming work will become more interesting and satisfying.

Running Python Programs from a Terminal

You'll run most of your programs directly in your text editor. However, sometimes it's useful to run programs from a terminal instead. For example, you might want to run an existing program without opening it for editing.

You can do this on any system with Python installed if you know how to access the directory where the program file is stored. To try this, make sure you've saved the *hello_world.py* file in the *python_work* folder on your desktop.

On Windows

You can use the terminal command `cd`, for *change directory*, to navigate through your filesystem in a command window. The command `dir`, for *directory*, shows you all the files that exist in the current directory.

Open a new terminal window and enter the following commands to run *hello_world.py*:

```
C:\> cd Desktop\python_work
C:\Desktop\python_work> dir
hello_world.py
C:\Desktop\python_work> python hello_world.py
Hello Python world!
```

First, use the **`cd`** command to navigate to the *python_work* folder, which is in the *Desktop* folder. Next, use the **`dir`** command to make sure *hello_world.py* is in this folder. Then run the file using the command **`python hello_world.py`**.

Most of your programs will run fine directly from your editor. However, as your work becomes more complex, you'll want to run some of your programs from a terminal.

On macOS and Linux

Running a Python program from a terminal session is the same on Linux and macOS. You can use the terminal command `cd`, for *change directory*, to navigate through your filesystem in a terminal session. The command `ls`, for *list*, shows you all the nonhidden files that exist in the current directory.

Open a new terminal window and enter the following commands to run *hello_world.py*:

```
~$ cd Desktop/python_work/
~/Desktop/python_work$ ls
hello_world.py
~/Desktop/python_work$ python3 hello_world.py
Hello Python world!
```

First, use the **`cd`** command to navigate to the *python_work* folder, which is in the *Desktop* folder. Next, use the **`ls`** command to make sure *hello_world.py* is in this folder. Then run the file using the command **`python3 hello_world.py`**.

Most of your programs will run fine directly from your editor. But as your work becomes more complex, you'll want to run some of your programs from a terminal.

TRY IT YOURSELF

The exercises in this chapter are exploratory in nature. Starting in Chapter 2, the challenges you'll solve will be based on what you've learned.

1-1. python.org: Explore the Python home page (*https://python.org*) to find topics that interest you. As you become familiar with Python, different parts of the site will be more useful to you.

1-2. Hello World Typos: Open the *hello_world.py* file you just created. Make a typo somewhere in the line and run the program again. Can you make a typo that generates an error? Can you make sense of the error message? Can you make a typo that doesn't generate an error? Why do you think it didn't make an error?

1-3. Infinite Skills: If you had infinite programming skills, what would you build? You're about to learn how to program. If you have an end goal in mind, you'll have an immediate use for your new skills; now is a great time to write brief descriptions of what you want to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. Take a few minutes now to describe three programs you want to create.

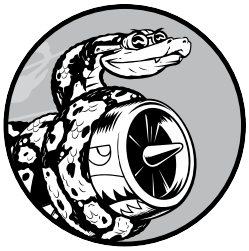
Summary

In this chapter, you learned a bit about Python in general, and you installed Python on your system if it wasn't already there. You also installed a text editor to make it easier to write Python code. You ran snippets of Python code in a terminal session, and you ran your first program, *hello_world.py*. You probably learned a bit about troubleshooting as well.

In the next chapter, you'll learn about the different kinds of data you can work with in your Python programs, and you'll start to use variables as well.

2

VARIABLES AND SIMPLE DATA TYPES



In this chapter you'll learn about the different kinds of data you can work with in your Python programs. You'll also learn how to use variables to represent data in your programs.

What Really Happens When You Run hello_world.py

Let's take a closer look at what Python does when you run *hello_world.py*. As it turns out, Python does a fair amount of work, even when it runs a simple program:

hello_world.py

```
print("Hello Python world!")
```

When you run this code, you should see the following output:

```
Hello Python world!
```

When you run the file *hello_world.py*, the ending *.py* indicates that the file is a Python program. Your editor then runs the file through the *Python interpreter*, which reads through the program and determines what each word in the program means. For example, when the interpreter sees the word `print` followed by parentheses, it prints to the screen whatever is inside the parentheses.

As you write your programs, your editor highlights different parts of your program in different ways. For example, it recognizes that `print()` is the name of a function and displays that word in one color. It recognizes that `"Hello Python world!"` is not Python code, and displays that phrase in a different color. This feature is called *syntax highlighting* and is quite useful as you start to write your own programs.

Variables

Let's try using a variable in *hello_world.py*. Add a new line at the beginning of the file, and modify the second line:

hello_world.py

```
message = "Hello Python world!"
print(message)
```

Run this program to see what happens. You should see the same output you saw previously:

```
Hello Python world!
```

We've added a *variable* named `message`. Every variable is connected to a *value*, which is the information associated with that variable. In this case the value is the `"Hello Python world!"` text.

Adding a variable makes a little more work for the Python interpreter. When it processes the first line, it associates the variable `message` with the `"Hello Python world!"` text. When it reaches the second line, it prints the value associated with `message` to the screen.

Let's expand on this program by modifying *hello_world.py* to print a second message. Add a blank line to *hello_world.py*, and then add two new lines of code:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

Now when you run *hello_world.py*, you should see two lines of output:

```
Hello Python world!
Hello Python Crash Course world!
```

You can change the value of a variable in your program at any time, and Python will always keep track of its current value.

Naming and Using Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand. Be sure to keep the following rules in mind when working with variables:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable `message_1` but not `1_message`.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, `greeting_message` works but `greeting message` will cause errors.
- Avoid using Python keywords and function names as variable names. For example, do not use the word `print` as a variable name; Python has reserved it for a particular programmatic purpose. (See "Python Keywords and Built-in Functions" on page 466.)
- Variable names should be short but descriptive. For example, `name` is better than `n`, `student_name` is better than `s_n`, and `name_length` is better than `length_of_persons_name`.
- Be careful when using the lowercase letter *l* and the uppercase letter *O* because they could be confused with the numbers *1* and *0*.

It can take some practice to learn how to create good variable names, especially as your programs become more interesting and complicated. As you write more programs and start to read through other people's code, you'll get better at coming up with meaningful names.

NOTE *The Python variables you're using at this point should be lowercase. You won't get errors if you use uppercase letters, but uppercase letters in variable names have special meanings that we'll discuss in later chapters.*

Avoiding Name Errors When Using Variables

Every programmer makes mistakes, and most make mistakes every day. Although good programmers might create errors, they also know how to respond to those errors efficiently. Let's look at an error you're likely to make early on and learn how to fix it.

We'll write some code that generates an error on purpose. Enter the following code, including the misspelled word `mesage`, which is shown in bold:

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

When an error occurs in your program, the Python interpreter does its best to help you figure out where the problem is. The interpreter provides a traceback when a program cannot run successfully. A *traceback* is a record of where the interpreter ran into trouble when trying to execute your code.

Here's an example of the traceback that Python provides after you've accidentally misspelled a variable's name:

```
Traceback (most recent call last):
❶  File "hello_world.py", line 2, in <module>
❷    print(mesage)
           ^^^^^^
❸ NameError: name 'mesage' is not defined. Did you mean: 'message'?
```

The output reports that an error occurs in line 2 of the file *hello_world.py* ❶. The interpreter shows this line ❷ to help us spot the error quickly and tells us what kind of error it found ❸. In this case it found a *name error* and reports that the variable being printed, `mesage`, has not been defined. Python can't identify the variable name provided. A name error usually means we either forgot to set a variable's value before using it, or we made a spelling mistake when entering the variable's name. If Python finds a variable name that's similar to the one it doesn't recognize, it will ask if that's the name you meant to use.

In this example we omitted the letter *s* in the variable name `message` in the second line. The Python interpreter doesn't spellcheck your code, but it does ensure that variable names are spelled consistently. For example, watch what happens when we spell *message* incorrectly in the line that defines the variable:

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

In this case, the program runs successfully!

```
Hello Python Crash Course reader!
```

The variable names match, so Python sees no issue. Programming languages are strict, but they disregard good and bad spelling. As a result, you don't need to consider English spelling and grammar rules when you're trying to create variable names and writing code.

Many programming errors are simple, single-character typos in one line of a program. If you find yourself spending a long time searching for one of these errors, know that you're in good company. Many experienced and talented programmers spend hours hunting down these kinds of tiny errors. Try to laugh about it and move on, knowing it will happen frequently throughout your programming life.

Variables Are Labels

Variables are often described as boxes you can store values in. This idea can be helpful the first few times you use a variable, but it isn't an accurate way to describe how variables are represented internally in Python. It's much better to think of variables as labels that you can assign to values. You can also say that a variable references a certain value.

This distinction probably won't matter much in your initial programs, but it's worth learning earlier rather than later. At some point, you'll see unexpected behavior from a variable, and an accurate understanding of how variables work will help you identify what's happening in your code.

NOTE *The best way to understand new programming concepts is to try using them in your programs. If you get stuck while working on an exercise in this book, try doing something else for a while. If you're still stuck, review the relevant part of that chapter. If you still need help, see the suggestions in Appendix C.*

TRY IT YOURSELF

Write a separate program to accomplish each of these exercises. Save each program with a filename that follows standard Python conventions, using lowercase letters and underscores, such as *simple_message.py* and *simple_messages.py*.

2-1. Simple Message: Assign a message to a variable, and then print that message.

2-2. Simple Messages: Assign a message to a variable, and print that message. Then change the value of the variable to a new message, and print the new message.

Strings

Because most programs define and gather some sort of data and then do something useful with it, it helps to classify different types of data. The first data type we'll look at is the string. Strings are quite simple at first glance, but you can use them in many different ways.

A *string* is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Let's explore some of the ways you can use strings.

Changing Case in a String with Methods

One of the simplest tasks you can do with strings is change the case of the words in a string. Look at the following code, and try to determine what's happening:

name.py

```
name = "ada lovelace"
print(name.title())
```

Save this file as *name.py* and then run it. You should see this output:

```
Ada Lovelace
```

In this example, the variable `name` refers to the lowercase string `"ada lovelace"`. The method `title()` appears after the variable in the `print()` call. A *method* is an action that Python can perform on a piece of data. The dot (`.`) after `name` in `name.title()` tells Python to make the `title()` method act on the variable `name`. Every method is followed by a set of parentheses, because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty.

The `title()` method changes each word to title case, where each word begins with a capital letter. This is useful because you'll often want to think of a name as a piece of information. For example, you might want your program to recognize the input values `Ada`, `ADA`, and `ada` as the same name, and display all of them as `Ada`.

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

This will display the following:

```
ADA LOVELACE
ada lovelace
```

The `lower()` method is particularly useful for storing data. You typically won't want to trust the capitalization that your users provide, so you'll convert strings to lowercase before storing them. Then when you want to display the information, you'll use the case that makes the most sense for each string.

Using Variables in Strings

In some situations, you'll want to use a variable's value inside a string. For example, you might want to use two variables to represent a first name and

a last name, respectively, and then combine those values to display someone's full name:

full_name.py

```
first_name = "ada"
last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
print(full_name)
```

To insert a variable's value into a string, place the letter `f` immediately before the opening quotation mark ❶. Put braces around the name or names of any variable you want to use inside the string. Python will replace each variable with its value when the string is displayed.

These strings are called *f-strings*. The *f* is for *format*, because Python formats the string by replacing the name of any variable in braces with its value. The output from the previous code is:

```
ada lovelace
```

You can do a lot with f-strings. For example, you can use f-strings to compose complete messages using the information associated with a variable, as shown here:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ print(f"Hello, {full_name.title()}!")
```

The full name is used in a sentence that greets the user ❶, and the `title()` method changes the name to title case. This code returns a simple but nicely formatted greeting:

```
Hello, Ada Lovelace!
```

You can also use f-strings to compose a message, and then assign the entire message to a variable:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ message = f"Hello, {full_name.title()}!"
❷ print(message)
```

This code displays the message `Hello, Ada Lovelace!` as well, but by assigning the message to a variable ❶ we make the final `print()` call much simpler ❷.

Adding Whitespace to Strings with Tabs or Newlines

In programming, *whitespace* refers to any nonprinting characters, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read.

To add a tab to your text, use the character combination \t:

```
>>> print("Python")
Python
>>> print("\tPython")
    Python
```

To add a newline in a string, use the character combination \n:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

You can also combine tabs and newlines in a single string. The string "\n\t" tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Newlines and tabs will be very useful in the next two chapters, when you start to produce many lines of output from just a few lines of code.

Stripping Whitespace

Extra whitespace can be confusing in your programs. To programmers, 'python' and 'python ' look pretty much the same. But to a program, they are two different strings. Python detects the extra space in 'python ' and considers it significant unless you tell it otherwise.

It's important to think about whitespace, because often you'll want to compare two strings to determine whether they are the same. For example, one important instance might involve checking people's usernames when they log in to a website. Extra whitespace can be confusing in much simpler situations as well. Fortunately, Python makes it easy to eliminate extra whitespace from data that people enter.

Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right side of a string, use the rstrip() method:

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language
  'python '
❸ >>> favorite_language.rstrip()
  'python'
❹ >>> favorite_language
  'python '
```

The value associated with `favorite_language` ❶ contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value ❷. When the `rstrip()` method acts on the variable `favorite_language` ❸, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, the string looks the same as when it was entered, including the extra whitespace ❹.

To remove the whitespace from the string permanently, you have to associate the stripped value with the variable name:

```
>>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then associate this new value with the original variable ❶. Changing a variable's value is done often in programming. This is how a variable's value can be updated as a program is executed or in response to user input.

You can also strip whitespace from the left side of a string using the `lstrip()` method, or from both sides at once using `strip()`:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
' python'
❸ >>> favorite_language.lstrip()
'python '
❹ >>> favorite_language.strip()
'python'
```

In this example, we start with a value that has whitespace at the beginning and the end ❶. We then remove the extra space from the right side ❷, from the left side ❸, and from both sides ❹. Experimenting with these stripping functions can help you become familiar with manipulating strings. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

Removing Prefixes

When working with strings, another common task is to remove a prefix. Consider a URL with the common prefix *https://*. We want to remove this prefix, so we can focus on just the part of the URL that users need to enter into an address bar. Here's how to do that:

```
>>> nostarch_url = 'https://nostarch.com'
>>> nostarch_url.removeprefix('https://')
'nostarch.com'
```

Enter the name of the variable followed by a dot, and then the method `removeprefix()`. Inside the parentheses, enter the prefix you want to remove from the original string.

Like the methods for removing whitespace, `removeprefix()` leaves the original string unchanged. If you want to keep the new value with the prefix removed, either reassign it to the original variable or assign it to a new variable:

```
>>> simple_url = nostarch_url.removeprefix('https://')
```

When you see a URL in an address bar and the *https://* part isn't shown, the browser is probably using a method like `removeprefix()` behind the scenes.

Avoiding Syntax Errors with Strings

One kind of error that you might see with some regularity is a syntax error. A *syntax error* occurs when Python doesn't recognize a section of your program as valid Python code. For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.

Here's how to use single and double quotes correctly. Save this program as *apostrophe.py* and then run it:

apostrophe.py

```
message = "One of Python's strengths is its diverse community."
print(message)
```

The apostrophe appears inside a set of double quotes, so the Python interpreter has no trouble reading the string correctly:

```
One of Python's strengths is its diverse community.
```

However, if you use single quotes, Python can't identify where the string should end:

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

You'll see the following output:

```
  File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
                                                                  ❶ ^
SyntaxError: unterminated string literal (detected at line 1)
```

In the output you can see that the error occurs right after the final single quote ❶. This syntax error indicates that the interpreter doesn't recognize

something in the code as valid Python code, and it thinks the problem might be a string that's not quoted correctly. Errors can come from a variety of sources, and I'll point out some common ones as they arise. You might see syntax errors often as you learn to write proper Python code. Syntax errors are also the least specific kind of error, so they can be difficult and frustrating to identify and correct. If you get stuck on a particularly stubborn error, see the suggestions in Appendix C.

NOTE *Your editor's syntax highlighting feature should help you spot some syntax errors quickly as you write your programs. If you see Python code highlighted as if it's English or English highlighted as if it's Python code, you probably have a mismatched quotation mark somewhere in your file.*

TRY IT YOURSELF

Save each of the following exercises as a separate file, with a name like *name_cases.py*. If you get stuck, take a break or see the suggestions in Appendix C.

2-3. Personal Message: Use a variable to represent a person's name, and print a message to that person. Your message should be simple, such as, "Hello Eric, would you like to learn some Python today?"

2-4. Name Cases: Use a variable to represent a person's name, and then print that person's name in lowercase, uppercase, and title case.

2-5. Famous Quote: Find a quote from a famous person you admire. Print the quote and the name of its author. Your output should look something like the following, including the quotation marks:

> *Albert Einstein once said, "A person who never made a mistake never tried anything new."*

2-6. Famous Quote 2: Repeat Exercise 2-5, but this time, represent the famous person's name using a variable called `famous_person`. Then compose your message and represent it with a new variable called `message`. Print your message.

2-7. Stripping Names: Use a variable to represent a person's name, and include some whitespace characters at the beginning and end of the name. Make sure you use each character combination, `"\t"` and `"\n"`, at least once.

Print the name once, so the whitespace around the name is displayed. Then print the name using each of the three stripping functions, `lstrip()`, `rstrip()`, and `strip()`.

2-8. File Extensions: Python has a `removesuffix()` method that works exactly like `removeprefix()`. Assign the value `'python_notes.txt'` to a variable called `filename`. Then use the `removesuffix()` method to display the filename without the file extension, like some file browsers do.

Numbers

Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on. Python treats numbers in several different ways, depending on how they're being used. Let's first look at how Python manages integers, because they're the simplest to work with.

Integers

You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify. For example:

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

The spacing in these examples has no effect on how Python evaluates the expressions; it simply helps you more quickly spot the operations that have priority when you're reading through the code.

Floats

Python calls any number with a decimal point a *float*. This term is used in most programming languages, and it refers to the fact that a decimal point

can appear at any position in a number. Every programming language must be carefully designed to properly manage decimal numbers so numbers behave appropriately, no matter where the decimal point appears.

For the most part, you can use floats without worrying about how they behave. Simply enter the numbers you want to use, and Python will most likely do what you expect:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

However, be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally. Just ignore the extra decimal places for now; you'll learn ways to deal with the extra places when you need to in the projects in Part II.

Integers and Floats

When you divide any two numbers, even if they are integers that result in a whole number, you'll always get a float:

```
>>> 4/2
2.0
```

If you mix an integer and a float in any other operation, you'll get a float as well:

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

Python defaults to a float in any operation that uses a float, even if the output is a whole number.

Underscores in Numbers

When you're writing long numbers, you can group digits using underscores to make large numbers more readable:

```
>>> universe_age = 14_000_000_000
```

When you print a number that was defined using underscores, Python prints only the digits:

```
>>> print(universe_age)
14000000000
```

Python ignores the underscores when storing these kinds of values. Even if you don't group the digits in threes, the value will still be unaffected. To Python, `1000` is the same as `1_000`, which is the same as `10_00`. This feature works for both integers and floats.

Multiple Assignment

You can assign values to more than one variable using just a single line of code. This can help shorten your programs and make them easier to read; you'll use this technique most often when initializing a set of numbers.

For example, here's how you can initialize the variables `x`, `y`, and `z` to zero:

```
>>> x, y, z = 0, 0, 0
```

You need to separate the variable names with commas, and do the same with the values, and Python will assign each value to its respective variable. As long as the number of values matches the number of variables, Python will match them up correctly.

Constants

A *constant* is a variable whose value stays the same throughout the life of a program. Python doesn't have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed:

```
MAX_CONNECTIONS = 5000
```

When you want to treat a variable as a constant in your code, write the name of the variable in all capital letters.

TRY IT YOURSELF

2-9. Number Eight: Write addition, subtraction, multiplication, and division operations that each result in the number 8. Be sure to enclose your operations in `print()` calls to see the results. You should create four lines that look like this:

```
print(5+3)
```

Your output should be four lines, with the number 8 appearing once on each line.

2-10. Favorite Number: Use a variable to represent your favorite number. Then, using that variable, create a message that reveals your favorite number. Print that message.

Comments

Comments are an extremely useful feature in most programming languages. Everything you've written in your programs so far is Python code. As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A *comment* allows you to write notes in your spoken language, within your programs.

How Do You Write Comments?

In Python, the hash mark (#) indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter. For example:

comment.py

```
# Say hello to everyone.
print("Hello Python people!")
```

Python ignores the first line and executes the second line.

```
Hello Python people!
```

What Kinds of Comments Should You Write?

The main reason to write comments is to explain what your code is supposed to do and how you are making it work. When you're in the middle of working on a project, you understand how all of the pieces fit together. But when you return to a project after some time away, you'll likely have

forgotten some of the details. You can always study your code for a while and figure out how segments were supposed to work, but writing good comments can save you time by summarizing your overall approach clearly.

If you want to become a professional programmer or collaborate with other programmers, you should write meaningful comments. Today, most software is written collaboratively, whether by a group of employees at one company or a group of people working together on an open source project. Skilled programmers expect to see comments in code, so it's best to start adding descriptive comments to your programs now. Writing clear, concise comments in your code is one of the most beneficial habits you can form as a new programmer.

When you're deciding whether to write a comment, ask yourself if you had to consider several approaches before coming up with a reasonable way to make something work; if so, write a comment about your solution. It's much easier to delete extra comments later than to go back and write comments for a sparsely commented program. From now on, I'll use comments in examples throughout this book to help explain sections of code.

TRY IT YOURSELF

2-11. Adding Comments: Choose two of the programs you've written, and add at least one comment to each. If you don't have anything specific to write because your programs are too simple at this point, just add your name and the current date at the top of each program file. Then write one sentence describing what the program does.

The Zen of Python

Experienced Python programmers will encourage you to avoid complexity and aim for simplicity whenever possible. The Python community's philosophy is contained in "The Zen of Python" by Tim Peters. You can access this brief set of principles for writing good Python code by entering `import this` into your interpreter. I won't reproduce the entire "Zen of Python" here, but I'll share a few lines to help you understand why they should be important to you as a beginning Python programmer.

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

Python programmers embrace the notion that code can be beautiful and elegant. In programming, people solve problems. Programmers have always respected well-designed, efficient, and even beautiful solutions to problems. As you learn more about Python and use it to write more code,

someone might look over your shoulder one day and say, "Wow, that's some beautiful code!"

```
Simple is better than complex.
```

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

```
Complex is better than complicated.
```

Real life is messy, and sometimes a simple solution to a problem is unattainable. In that case, use the simplest solution that works.

```
Readability counts.
```

Even when your code is complex, aim to make it readable. When you're working on a project that involves complex coding, focus on writing informative comments for that code.

```
There should be one-- and preferably only one --obvious way to do it.
```

If two Python programmers are asked to solve the same problem, they should come up with fairly compatible solutions. This is not to say there's no room for creativity in programming. On the contrary, there is plenty of room for creativity! However, much of programming consists of using small, common approaches to simple situations within a larger, more creative project. The nuts and bolts of your programs should make sense to other Python programmers.

```
Now is better than never.
```

You could spend the rest of your life learning all the intricacies of Python and of programming in general, but then you'd never complete any projects. Don't try to write perfect code; write code that works, and then decide whether to improve your code for that project or move on to something new.

As you continue to the next chapter and start digging into more involved topics, try to keep this philosophy of simplicity and clarity in mind. Experienced programmers will respect your code more and will be happy to give you feedback and collaborate with you on interesting projects.

TRY IT YOURSELF

2-12. Zen of Python: Enter `import this` into a Python terminal session and skim through the additional principles.

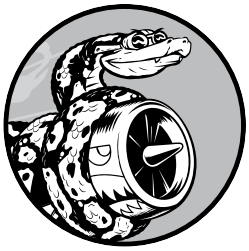
Summary

In this chapter you learned how to work with variables. You learned to use descriptive variable names and resolve name errors and syntax errors when they arise. You learned what strings are and how to display them using lowercase, uppercase, and title case. You started using whitespace to organize output neatly, and you learned how to remove unneeded elements from a string. You started working with integers and floats, and you learned some of the ways you can work with numerical data. You also learned to write explanatory comments to make your code easier for you and others to read. Finally, you read about the philosophy of keeping your code as simple as possible, whenever possible.

In Chapter 3, you'll learn how to store collections of information in data structures called *lists*. You'll also learn how to work through a list, manipulating any information in that list.

3

INTRODUCING LISTS



In this chapter and the next you'll learn what lists are and how to start working with the elements in a list. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

What Is a List?

A *list* is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0 to 9, or the names of all the people in your family. You can put anything you want into a list, and the items in your list don't have to be related in any particular way. Because

a list usually contains more than one element, it's a good idea to make the name of your list plural, such as `letters`, `digits`, or `names`.

In Python, square brackets ([]) indicate a list, and individual elements in the list are separated by commas. Here's a simple example of a list that contains a few kinds of bicycles:

bicycles.py

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

If you ask Python to print a list, Python returns its representation of the list, including the square brackets:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Because this isn't the output you want your users to see, let's learn how to access the individual items in a list.

Accessing Elements in a List

Lists are ordered collections, so you can access any element in a list by telling Python the position, or *index*, of the item desired. To access an element in a list, write the name of the list followed by the index of the item enclosed in square brackets.

For example, let's pull out the first bicycle in the list `bicycles`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
```

When we ask for a single item from a list, Python returns just that element without square brackets:

```
trek
```

This is the result you want your users to see: clean, neatly formatted output.

You can also use the string methods from Chapter 2 on any element in this list. For example, you can format the element `'trek'` to look more presentable by using the `title()` method:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

This example produces the same output as the preceding example, except `'Trek'` is capitalized.

Index Positions Start at 0, Not 1

Python considers the first item in a list to be at position 0, not position 1. This is true of most programming languages, and the reason has to do with

how the list operations are implemented at a lower level. If you're receiving unexpected results, ask yourself if you're making a simple but common off-by-one error.

The second item in a list has an index of 1. Using this counting system, you can get any element you want from a list by subtracting one from its position in the list. For instance, to access the fourth item in a list, you request the item at index 3.

The following asks for the bicycles at index `1` and index `3`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

This code returns the second and fourth bicycles in the list:

```
cannondale
specialized
```

Python has a special syntax for accessing the last element in a list. If you ask for the item at index `-1`, Python always returns the last item in the list:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

This code returns the value `'specialized'`. This syntax is quite useful, because you'll often want to access the last items in a list without knowing exactly how long the list is. This convention extends to other negative index values as well. The index `-2` returns the second item from the end of the list, the index `-3` returns the third item from the end, and so forth.

Using Individual Values from a List

You can use individual values from a list just as you would any other variable. For example, you can use f-strings to create a message based on a value from a list.

Let's try pulling the first bicycle from the list and composing a message using that value:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
message = f"My first bicycle was a {bicycles[0].title()}."

print(message)
```

We build a sentence using the value at `bicycles[0]` and assign it to the variable `message`. The output is a simple sentence about the first bicycle in the list:

```
My first bicycle was a Trek.
```

TRY IT YOURSELF

Try these short programs to get some firsthand experience with Python's lists. You might want to create a new folder for each chapter's exercises, to keep them organized.

3-1. Names: Store the names of a few of your friends in a list called `names`. Print each person's name by accessing each element in the list, one at a time.

3-2. Greetings: Start with the list you used in Exercise 3-1, but instead of just printing each person's name, print a message to them. The text of each message should be the same, but each message should be personalized with the person's name.

3-3. Your Own List: Think of your favorite mode of transportation, such as a motorcycle or a car, and make a list that stores several examples. Use your list to print a series of statements about these items, such as "I would like to own a Honda motorcycle."

Modifying, Adding, and Removing Elements

Most lists you create will be *dynamic*, meaning you'll build a list and then add and remove elements from it as your program runs its course. For example, you might create a game in which a player has to shoot aliens out of the sky. You could store the initial set of aliens in a list and then remove an alien from the list each time one is shot down. Each time a new alien appears on the screen, you add it to the list. Your list of aliens will increase and decrease in length throughout the course of the game.

Modifying Elements in a List

The syntax for modifying an element is similar to the syntax for accessing an element in a list. To change an element, use the name of the list followed by the index of the element you want to change, and then provide the new value you want that item to have.

For example, say we have a list of motorcycles and the first item in the list is `'honda'`. We can change the value of this first item after the list has been created:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles[0] = 'ducati'
print(motorcycles)
```

Here we define the list `motorcycles`, with `'honda'` as the first element. Then we change the value of the first item to `'ducati'`. The output shows that the first item has been changed, while the rest of the list stays the same:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

You can change the value of any item in a list, not just the first item.

Adding Elements to a List

You might want to add a new element to a list for many reasons. For example, you might want to make new aliens appear in a game, add new data to a visualization, or add new registered users to a website you've built. Python provides several ways to add new data to existing lists.

Appending Elements to the End of a List

The simplest way to add a new element to a list is to *append* the item to the list. When you append an item to a list, the new element is added to the end of the list. Using the same list we had in the previous example, we'll add the new element `'ducati'` to the end of the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles.append('ducati')
print(motorcycles)
```

Here the `append()` method adds `'ducati'` to the end of the list, without affecting any of the other elements in the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

The `append()` method makes it easy to build lists dynamically. For example, you can start with an empty list and then add items to the list using a series of `append()` calls. Using an empty list, let's add the elements `'honda'`, `'yamaha'`, and `'suzuki'` to the list:

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

The resulting list looks exactly the same as the lists in the previous examples:

```
['honda', 'yamaha', 'suzuki']
```

Building lists this way is very common, because you often won't know the data your users want to store in a program until after the program is running. To put your users in control, start by defining an empty list that will hold the users' values. Then append each new value provided to the list you just created.

Inserting Elements into a List

You can add a new element at any position in your list by using the `insert()` method. You do this by specifying the index of the new element and the value of the new item:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

motorcycles.insert(0, 'ducati')
print(motorcycles)
```

In this example, we insert the value `'ducati'` at the beginning of the list. The `insert()` method opens a space at position `0` and stores the value `'ducati'` at that location:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

This operation shifts every other value in the list one position to the right.

Removing Elements from a List

Often, you'll want to remove an item or a set of items from a list. For example, when a player shoots down an alien from the sky, you'll most likely want to remove it from the list of active aliens. Or when a user decides to cancel their account on a web application you created, you'll want to remove that user from the list of active users. You can remove an item according to its position in the list or according to its value.

Removing an Item Using the del Statement

If you know the position of the item you want to remove from a list, you can use the `del` statement:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[0]
print(motorcycles)
```

Here we use the `del` statement to remove the first item, `'honda'`, from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

You can remove an item from any position in a list using the `del` statement if you know its index. For example, here's how to remove the second item, `'yamaha'`, from the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

The second motorcycle is deleted from the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

In both examples, you can no longer access the value that was removed from the list after the `del` statement is used.

Removing an Item Using the pop() Method

Sometimes you'll want to use the value of an item after you remove it from a list. For example, you might want to get the *x* and *y* position of an alien that was just shot down, so you can draw an explosion at that position. In a web application, you might want to remove a user from a list of active members and then add that user to a list of inactive members.

The `pop()` method removes the last item in a list, but it lets you work with that item after removing it. The term *pop* comes from thinking of a list as a stack of items and popping one item off the top of the stack. In this analogy, the top of a stack corresponds to the end of a list.

Let's pop a motorcycle from the list of motorcycles:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

We start by defining and printing the list `motorcycles` ❶. Then we pop a value from the list, and assign that value to the variable `popped_motorcycle` ❷. We print the list ❸ to show that a value has been removed from the list. Then we print the popped value ❹ to prove that we still have access to the value that was removed.

The output shows that the value `'suzuki'` was removed from the end of the list and is now assigned to the variable `popped_motorcycle`:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

How might this `pop()` method be useful? Imagine that the motorcycles in the list are stored in chronological order, according to when we owned them. If this is the case, we can use the `pop()` method to print a statement about the last motorcycle we bought:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

The output is a simple sentence about the most recent motorcycle we owned:

```
The last motorcycle I owned was a Suzuki.
```

Popping Items from Any Position in a List

You can use `pop()` to remove an item from any position in a list by including the index of the item you want to remove in parentheses:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

first_owned = motorcycles.pop(0)
print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

We start by popping the first motorcycle in the list, and then we print a message about that motorcycle. The output is a simple sentence describing the first motorcycle I ever owned:

```
The first motorcycle I owned was a Honda.
```

Remember that each time you use `pop()`, the item you work with is no longer stored in the list.

If you're unsure whether to use the `del` statement or the `pop()` method, here's a simple way to decide: when you want to delete an item from a list and not use that item in any way, use the `del` statement; if you want to use an item as you remove it, use the `pop()` method.

Removing an Item by Value

Sometimes you won't know the position of the value you want to remove from a list. If you only know the value of the item you want to remove, you can use the `remove()` method.

For example, say we want to remove the value `'ducati'` from the list of motorcycles:

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

motorcycles.remove('ducati')
print(motorcycles)
```

Here the `remove()` method tells Python to figure out where `'ducati'` appears in the list and remove that element:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

You can also use the `remove()` method to work with a value that's being removed from a list. Let's remove the value `'ducati'` and print a reason for removing it from the list:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
  print(motorcycles)

❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
  print(motorcycles)
❹ print(f"\nA {too_expensive.title()} is too expensive for me.")
```

After defining the list ❶, we assign the value `'ducati'` to a variable called `too_expensive` ❷. We then use this variable to tell Python which value to remove from the list ❸. The value `'ducati'` has been removed from the list ❹ but is still accessible through the variable `too_expensive`, allowing us to print a statement about why we removed `'ducati'` from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']

A Ducati is too expensive for me.
```

NOTE *The `remove()` method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to make sure all occurrences of the value are removed. You'll learn how to do this in Chapter 7.*

TRY IT YOURSELF

The following exercises are a bit more complex than those in Chapter 2, but they give you an opportunity to use lists in all of the ways described.

3-4. Guest List: If you could invite anyone, living or deceased, to dinner, who would you invite? Make a list that includes at least three people you'd like to invite to dinner. Then use your list to print a message to each person, inviting them to dinner.

(continued)

3-5. Changing Guest List: You just heard that one of your guests can't make the dinner, so you need to send out a new set of invitations. You'll have to think of someone else to invite.

- Start with your program from Exercise 3-4. Add a `print()` call at the end of your program, stating the name of the guest who can't make it.
- Modify your list, replacing the name of the guest who can't make it with the name of the new person you are inviting.
- Print a second set of invitation messages, one for each person who is still in your list.

3-6. More Guests: You just found a bigger dinner table, so now more space is available. Think of three more guests to invite to dinner.

- Start with your program from Exercise 3-4 or 3-5. Add a `print()` call to the end of your program, informing people that you found a bigger table.
- Use `insert()` to add one new guest to the beginning of your list.
- Use `insert()` to add one new guest to the middle of your list.
- Use `append()` to add one new guest to the end of your list.
- Print a new set of invitation messages, one for each person in your list.

3-7. Shrinking Guest List: You just found out that your new dinner table won't arrive in time for the dinner, and now you have space for only two guests.

- Start with your program from Exercise 3-6. Add a new line that prints a message saying that you can invite only two people for dinner.
- Use `pop()` to remove guests from your list one at a time until only two names remain in your list. Each time you pop a name from your list, print a message to that person letting them know you're sorry you can't invite them to dinner.
- Print a message to each of the two people still on your list, letting them know they're still invited.
- Use `del` to remove the last two names from your list, so you have an empty list. Print your list to make sure you actually have an empty list at the end of your program.

Organizing a List

Often, your lists will be created in an unpredictable order, because you can't always control the order in which your users provide their data. Although this is unavoidable in most circumstances, you'll frequently want to present your information in a particular order. Sometimes you'll want

to preserve the original order of your list, and other times you'll want to change the original order. Python provides a number of different ways to organize your lists, depending on the situation.

Sorting a List Permanently with the sort() Method

Python's `sort()` method makes it relatively easy to sort a list. Imagine we have a list of cars and want to change the order of the list to store them alphabetically. To keep the task simple, let's assume that all the values in the list are lowercase:

cars.py

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort()
print(cars)
```

The `sort()` method changes the order of the list permanently. The cars are now in alphabetical order, and we can never revert to the original order:

```
['audi', 'bmw', 'subaru', 'toyota']
```

You can also sort this list in reverse-alphabetical order by passing the argument `reverse=True` to the `sort()` method. The following example sorts the list of cars in reverse-alphabetical order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

Again, the order of the list is permanently changed:

```
['toyota', 'subaru', 'bmw', 'audi']
```

Sorting a List Temporarily with the sorted() Function

To maintain the original order of a list but present it in a sorted order, you can use the `sorted()` function. The `sorted()` function lets you display your list in a particular order, but doesn't affect the actual order of the list.

Let's try this function on the list of cars.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

❶ print("Here is the original list:")
  print(cars)

❷ print("\nHere is the sorted list:")
  print(sorted(cars))

❸ print("\nHere is the original list again:")
  print(cars)
```

We first print the list in its original order ❶ and then in alphabetical order ❷. After the list is displayed in the new order, we show that the list is still stored in its original order ❸:

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']

Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']

❶ Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']
```

Notice that the list still exists in its original order ❶ after the `sorted()` function has been used. The `sorted()` function can also accept a `reverse=True` argument if you want to display a list in reverse-alphabetical order.

NOTE *Sorting a list alphabetically is a bit more complicated when all the values are not in lowercase. There are several ways to interpret capital letters when determining a sort order, and specifying the exact order can be more complex than we want to deal with at this time. However, most approaches to sorting will build directly on what you learned in this section.*

Printing a List in Reverse Order

To reverse the original order of a list, you can use the `reverse()` method. If we originally stored the list of cars in chronological order according to when we owned them, we could easily rearrange the list into reverse-chronological order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

Notice that `reverse()` doesn't sort backward alphabetically; it simply reverses the order of the list:

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

The `reverse()` method changes the order of a list permanently, but you can revert to the original order anytime by applying `reverse()` to the same list a second time.

Finding the Length of a List

You can quickly find the length of a list by using the `len()` function. The list in this example has four items, so its length is `4`:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

You'll find `len()` useful when you need to identify the number of aliens that still need to be shot down in a game, determine the amount of data you have to manage in a visualization, or figure out the number of registered users on a website, among other tasks.

NOTE *Python counts the items in a list starting with one, so you shouldn't run into any off-by-one errors when determining the length of a list.*

TRY IT YOURSELF

3-8. Seeing the World: Think of at least five places in the world you'd like to visit.

- Store the locations in a list. Make sure the list is not in alphabetical order.
- Print your list in its original order. Don't worry about printing the list neatly; just print it as a raw Python list.
- Use `sorted()` to print your list in alphabetical order without modifying the actual list.
- Show that your list is still in its original order by printing it.
- Use `sorted()` to print your list in reverse-alphabetical order without changing the order of the original list.
- Show that your list is still in its original order by printing it again.
- Use `reverse()` to change the order of your list. Print the list to show that its order has changed.
- Use `reverse()` to change the order of your list again. Print the list to show it's back to its original order.
- Use `sort()` to change your list so it's stored in alphabetical order. Print the list to show that its order has been changed.
- Use `sort()` to change your list so it's stored in reverse-alphabetical order. Print the list to show that its order has changed.

3-9. Dinner Guests: Working with one of the programs from Exercises 3-4 through 3-7 (pages 41–42), use `len()` to print a message indicating the number of people you're inviting to dinner.

3-10. Every Function: Think of things you could store in a list. For example, you could make a list of mountains, rivers, countries, cities, languages, or anything else you'd like. Write a program that creates a list containing these items and then uses each function introduced in this chapter at least once.

Avoiding Index Errors When Working with Lists

There's one type of error that's common to see when you're working with lists for the first time. Let's say you have a list with three items, and you ask for the fourth item:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

This example results in an *index error*:

```
Traceback (most recent call last):
  File "motorcycles.py", line 2, in <module>
    print(motorcycles[3])
          ~~~~~~~~~~~^^^
IndexError: list index out of range
```

Python attempts to give you the item at index 3. But when it searches the list, no item in `motorcycles` has an index of 3. Because of the off-by-one nature of indexing in lists, this error is typical. People think the third item is item number 3, because they start counting at 1. But in Python the third item is number 2, because it starts indexing at 0.

An index error means Python can't find an item at the index you requested. If an index error occurs in your program, try adjusting the index you're asking for by one. Then run the program again to see if the results are correct.

Keep in mind that whenever you want to access the last item in a list, you should use the index `-1`. This will always work, even if your list has changed size since the last time you accessed it:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

The index `-1` always returns the last item in a list, in this case the value `'suzuki'`:

```
suzuki
```

The only time this approach will cause an error is when you request the last item from an empty list:

```
motorcycles = []
print(motorcycles[-1])
```

No items are in `motorcycles`, so Python returns another index error:

```
Traceback (most recent call last):
  File "motorcyles.py", line 3, in <module>
    print(motorcycles[-1])
          ~~~~~~~~~~~^^^^
IndexError: list index out of range
```

If an index error occurs and you can't figure out how to resolve it, try printing your list or just printing the length of your list. Your list might look much different than you thought it did, especially if it has been managed dynamically by your program. Seeing the actual list, or the exact number of items in your list, can help you sort out such logical errors.

TRY IT YOURSELF

3-11. Intentional Error: If you haven't received an index error in one of your programs yet, try to make one happen. Change an index in one of your programs to produce an index error. Make sure you correct the error before closing the program.

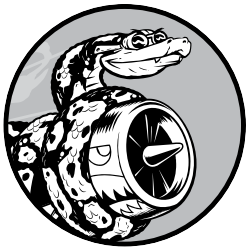
Summary

In this chapter, you learned what lists are and how to work with the individual items in a list. You learned how to define a list and how to add and remove elements. You learned how to sort lists permanently and temporarily for display purposes. You also learned how to find the length of a list and how to avoid index errors when you're working with lists.

In Chapter 4 you'll learn how to work with items in a list more efficiently. By looping through each item in a list using just a few lines of code you'll be able to work efficiently, even when your list contains thousands or millions of items.

4

WORKING WITH LISTS



In Chapter 3 you learned how to make a simple list, and you learned to work with the individual elements in a list. In this chapter you'll learn how to loop through an entire list using just a few lines of code, regardless of how long the list is. *Looping* allows you to take the same action, or set of actions, with every item in a list. As a result, you'll be able to work efficiently with lists of any length, including those with thousands or even millions of items.

Looping Through an Entire List

You'll often want to run through all entries in a list, performing the same task with each item. For example, in a game you might want to move every element on the screen by the same amount. In a list of numbers, you might want to perform the same statistical operation on every element.

Or perhaps you'll want to display each headline from a list of articles on a website. When you want to do the same action with every item in a list, you can use Python's `for` loop.

Say we have a list of magicians' names, and we want to print out each name in the list. We could do this by retrieving each name from the list individually, but this approach could cause several problems. For one, it would be repetitive to do this with a long list of names. Also, we'd have to change our code each time the list's length changed. Using a `for` loop avoids both of these issues by letting Python manage these issues internally.

Let's use a `for` loop to print out each name in a list of magicians:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician)
```

We begin by defining a list, just as we did in Chapter 3. Then we define a `for` loop. This line tells Python to pull a name from the list `magicians`, and associate it with the variable `magician`. Next, we tell Python to print the name that's just been assigned to `magician`. Python then repeats these last two lines, once for each name in the list. It might help to read this code as "For every magician in the list of magicians, print the magician's name." The output is a simple printout of each name in the list:

```
alice
david
carolina
```

A Closer Look at Looping

Looping is important because it's one of the most common ways a computer automates repetitive tasks. For example, in a simple loop like we used in *magicians.py*, Python initially reads the first line of the loop:

```
for magician in magicians:
```

This line tells Python to retrieve the first value from the list `magicians` and associate it with the variable `magician`. This first value is `'alice'`. Python then reads the next line:

```
    print(magician)
```

Python prints the current value of `magician`, which is still `'alice'`. Because the list contains more values, Python returns to the first line of the loop:

```
for magician in magicians:
```

Python retrieves the next name in the list, `'david'`, and associates that value with the variable `magician`. Python then executes the line:

```
    print(magician)
```

Python prints the current value of `magician` again, which is now `'david'`. Python repeats the entire loop once more with the last value in the list, `'carolina'`. Because no more values are in the list, Python moves on to the next line in the program. In this case nothing comes after the `for` loop, so the program ends.

When you're using loops for the first time, keep in mind that the set of steps is repeated once for each item in the list, no matter how many items are in the list. If you have a million items in your list, Python repeats these steps a million times—and usually very quickly.

Also keep in mind when writing your own `for` loops that you can choose any name you want for the temporary variable that will be associated with each value in the list. However, it's helpful to choose a meaningful name that represents a single item from the list. For example, here's a good way to start a `for` loop for a list of cats, a list of dogs, and a general list of items:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

These naming conventions can help you follow the action being done on each item within a `for` loop. Using singular and plural names can help you identify whether a section of code is working with a single element from the list or the entire list.

Doing More Work Within a for Loop

You can do just about anything with each item in a `for` loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
```

The only difference in this code is where we compose a message to each magician, starting with that magician's name. The first time through the loop the value of `magician` is `'alice'`, so Python starts the first message with the name `'Alice'`. The second time through, the message will begin with `'David'`, and the third time through, the message will begin with `'Carolina'`.

The output shows a personalized message for each magician in the list:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

You can also write as many lines of code as you like in the `for` loop. Every indented line following the line `for magician in magicians` is considered *inside the loop*, and each indented line is executed once for each value in the list. Therefore, you can do as much work as you like with each value in the list.

Let's add a second line to our message, telling each magician that we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

Because we have indented both calls to `print()`, each line will be executed once for every magician in the list. The newline (`"\n"`) in the second `print()` call inserts a blank line after each pass through the loop. This creates a set of messages that are neatly grouped for each person in the list:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

You can use as many lines as you like in your `for` loops. In practice, you'll often find it useful to do a number of different operations with each item in a list when you use a `for` loop.

Doing Something After a for Loop

What happens once a `for` loop has finished executing? Usually, you'll want to summarize a block of output or move on to other work that your program must accomplish.

Any lines of code after the `for` loop that are not indented are executed once without repetition. Let's write a thank you to the group of magicians as a whole, thanking them for putting on an excellent show. To display this group message after all of the individual messages have been printed, we place the thank you message after the `for` loop, without indentation:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

print("Thank you, everyone. That was a great magic show!")
```

The first two calls to `print()` are repeated once for each magician in the list, as you saw earlier. However, because the last line is not indented, it's printed only once:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
```

```
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

When you're processing data using a `for` loop, you'll find that this is a good way to summarize an operation that was performed on an entire dataset. For example, you might use a `for` loop to initialize a game by running through a list of characters and displaying each character on the screen. You might then write some additional code after this loop that displays a *Play Now* button after all the characters have been drawn to the screen.

Avoiding Indentation Errors

Python uses indentation to determine how a line, or group of lines, is related to the rest of the program. In the previous examples, the lines that printed messages to individual magicians were part of the `for` loop because they were indented. Python's use of indentation makes code very easy to read. Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure. In longer Python programs, you'll notice blocks of code indented at a few different levels. These indentation levels help you gain a general sense of the overall program's organization.

As you begin to write code that relies on proper indentation, you'll need to watch for a few common *indentation errors*. For example, people sometimes indent lines of code that don't need to be indented or forget to indent lines that need to be indented. Seeing examples of these errors now will help you avoid them in the future and correct them when they do appear in your own programs.

Let's examine some of the more common indentation errors.

Forgetting to Indent

Always indent the line after the `for` statement in a loop. If you forget, Python will remind you:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶ print(magician)
```

The call to `print()` ❶ should be indented, but it's not. When Python expects an indented block and doesn't find one, it lets you know which line it had a problem with:

```
  File "magicians.py", line 3
    print(magician)
    ^
IndentationError: expected an indented block after 'for' statement on line 2
```

You can usually resolve this kind of indentation error by indenting the line or lines immediately after the `for` statement.

Forgetting to Indent Additional Lines

Sometimes your loop will run without any errors but won't produce the expected result. This can happen when you're trying to do several tasks in a loop and you forget to indent some of its lines.

For example, this is what happens when we forget to indent the second line in the loop that tells each magician we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶ print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

The second call to `print()` ❶ is supposed to be indented, but because Python finds at least one indented line after the `for` statement, it doesn't report an error. As a result, the first `print()` call is executed once for each name in the list because it is indented. The second `print()` call is not indented, so it is executed only once after the loop has finished running. Because the final value associated with `magician` is `'carolina'`, she is the only one who receives the "looking forward to the next trick" message:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

This is a *logical error*. The syntax is valid Python code, but the code does not produce the desired result because a problem occurs in its logic. If you expect to see a certain action repeated once for each item in a list and it's executed only once, determine whether you need to simply indent a line or a group of lines.

Indenting Unnecessarily

If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

hello_world.py

```
message = "Hello Python world!"
    print(message)
```

We don't need to indent the `print()` call, because it isn't part of a loop; hence, Python reports that error:

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

You can avoid unexpected indentation errors by indenting only when you have a specific reason to do so. In the programs you're writing at this point, the only lines you should indent are the actions you want to repeat for each item in a `for` loop.

Indenting Unnecessarily After the Loop

If you accidentally indent code that should run after a loop has finished, that code will be repeated once for each item in the list. Sometimes this prompts Python to report an error, but often this will result in a logical error.

For example, let's see what happens when we accidentally indent the line that thanked the magicians as a group for putting on a good show:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

❶     print("Thank you everyone, that was a great magic show!")
```

Because the last line ❶ is indented, it's printed once for each person in the list:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you everyone, that was a great magic show!
```

This is another logical error, similar to the one in "Forgetting to Indent Additional Lines" on page 54. Because Python doesn't know what you're trying to accomplish with your code, it will run all code that is written in valid syntax. If an action is repeated many times when it should be executed only once, you probably need to unindent the code for that action.

Forgetting the Colon

The colon at the end of a `for` statement tells Python to interpret the next line as the start of a loop.

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians
    print(magician)
```

If you accidentally forget the colon ❶, you'll get a syntax error because Python doesn't know exactly what you're trying to do:

```
  File "magicians.py", line 2
    for magician in magicians
                             ^
SyntaxError: expected ':'
```

Python doesn't know if you simply forgot the colon, or if you meant to write additional code to set up a more complex loop. If the interpreter can identify a possible fix it will suggest one, like adding a colon at the end of a line, as it does here with the response `expected ':'`. Some errors have easy, obvious fixes, thanks to the suggestions in Python's tracebacks. Some errors are much harder to resolve, even when the eventual fix only involves a single character. Don't feel bad when a small fix takes a long time to find; you are absolutely not alone in this experience.

TRY IT YOURSELF

4-1. Pizzas: Think of at least three kinds of your favorite pizza. Store these pizza names in a list, and then use a `for` loop to print the name of each pizza.

- Modify your `for` loop to print a sentence using the name of the pizza, instead of printing just the name of the pizza. For each pizza, you should have one line of output containing a simple statement like *I like pepperoni pizza*.
- Add a line at the end of your program, outside the `for` loop, that states how much you like pizza. The output should consist of three or more lines about the kinds of pizza you like and then an additional sentence, such as *I really love pizza!*

4-2. Animals: Think of at least three different animals that have a common characteristic. Store the names of these animals in a list, and then use a `for` loop to print out the name of each animal.

- Modify your program to print a statement about each animal, such as *A dog would make a great pet.*
- Add a line at the end of your program, stating what these animals have in common. You could print a sentence, such as *Any of these animals would make a great pet!*

Making Numerical Lists

Many reasons exist to store a set of numbers. For example, you'll need to keep track of the positions of each character in a game, and you might want

to keep track of a player's high scores as well. In data visualizations, you'll almost always work with sets of numbers, such as temperatures, distances, population sizes, or latitude and longitude values, among other types of numerical sets.

Lists are ideal for storing sets of numbers, and Python provides a variety of tools to help you work efficiently with lists of numbers. Once you understand how to use these tools effectively, your code will work well even when your lists contain millions of items.

Using the range() Function

Python's `range()` function makes it easy to generate a series of numbers. For example, you can use the `range()` function to print a series of numbers like this:

first_numbers.py

```
for value in range(1, 5):
    print(value)
```

Although this code looks like it should print the numbers from 1 to 5, it doesn't print the number 5:

```
1
2
3
4
```

In this example, `range()` prints only the numbers 1 through 4. This is another result of the off-by-one behavior you'll see often in programming languages. The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide. Because it stops at that second value, the output never contains the end value, which would have been 5 in this case.

To print the numbers from 1 to 5, you would use `range(1, 6)`:

```
for value in range(1, 6):
    print(value)
```

This time the output starts at 1 and ends at 5:

```
1
2
3
4
5
```

If your output is different from what you expect when you're using `range()`, try adjusting your end value by 1.

You can also pass `range()` only one argument, and it will start the sequence of numbers at 0. For example, `range(6)` would return the numbers from 0 through 5.

Using range() to Make a List of Numbers

If you want to make a list of numbers, you can convert the results of `range()` directly into a list using the `list()` function. When you wrap `list()` around a call to the `range()` function, the output will be a list of numbers.

In the example in the previous section, we simply printed out a series of numbers. We can use `list()` to convert that same set of numbers into a list:

```
numbers = list(range(1, 6))
print(numbers)
```

This is the result:

```
[1, 2, 3, 4, 5]
```

We can also use the `range()` function to tell Python to skip numbers in a given range. If you pass a third argument to `range()`, Python uses that value as a step size when generating numbers.

For example, here's how to list the even numbers between 1 and 10:

even_numbers.py

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

In this example, the `range()` function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

```
[2, 4, 6, 8, 10]
```

You can create almost any set of numbers you want to using the `range()` function. For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10). In Python, two asterisks (**) represent exponents. Here's how you might put the first 10 square numbers into a list:

square_numbers.py

```
squares = []
for value in range(1, 11):
❶    square = value ** 2
❷    squares.append(square)

print(squares)
```

We start with an empty list called `squares`. Then, we tell Python to loop through each value from 1 to 10 using the `range()` function. Inside the loop, the current value is raised to the second power and assigned to the variable `square` ❶. Each new value of `square` is then appended to the list `squares` ❷. Finally, when the loop has finished running, the list of squares is printed:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To write this code more concisely, omit the temporary variable `square` and append each new value directly to the list:

```
squares = []
for value in range(1,11):
    squares.append(value**2)

print(squares)
```

This line does the same work as the lines inside the `for` loop in the previous listing. Each value in the loop is raised to the second power and then immediately appended to the list of squares.

You can use either of these approaches when you're making more complex lists. Sometimes using a temporary variable makes your code easier to read; other times it makes the code unnecessarily long. Focus first on writing code that you understand clearly, and does what you want it to do. Then look for more efficient approaches as you review your code.

Simple Statistics with a List of Numbers

A few Python functions are helpful when working with lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

NOTE *The examples in this section use short lists of numbers that fit easily on the page. They would work just as well if your list contained a million or more numbers.*

List Comprehensions

The approach described earlier for generating the list `squares` consisted of using three or four lines of code. A *list comprehension* allows you to generate this same list in just one line of code. A list comprehension combines the `for` loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but I've included them here because you'll most likely see them as soon as you start looking at other people's code.

The following example builds the same list of square numbers you saw earlier but uses a list comprehension:

squares.py

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

To use this syntax, begin with a descriptive name for the list, such as `squares`. Next, open a set of square brackets and define the expression for the values you want to store in the new list. In this example the expression is `value**2`, which raises the value to the second power. Then, write a `for` loop to generate the numbers you want to feed into the expression, and close the square brackets. The `for` loop in this example is `for value in range(1, 11)`, which feeds the values 1 through 10 into the expression `value**2`. Note that no colon is used at the end of the `for` statement.

The result is the same list of square numbers you saw earlier:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

It takes practice to write your own list comprehensions, but you'll find them worthwhile once you become comfortable creating ordinary lists. When you're writing three or four lines of code to generate lists and it begins to feel repetitive, consider writing your own list comprehensions.

TRY IT YOURSELF

4-3. Counting to Twenty: Use a `for` loop to print the numbers from 1 to 20, inclusive.

4-4. One Million: Make a list of the numbers from one to one million, and then use a `for` loop to print the numbers. (If the output is taking too long, stop it by pressing CTRL-C or by closing the output window.)

4-5. Summing a Million: Make a list of the numbers from one to one million, and then use `min()` and `max()` to make sure your list actually starts at one and ends at one million. Also, use the `sum()` function to see how quickly Python can add a million numbers.

4-6. Odd Numbers: Use the third argument of the `range()` function to make a list of the odd numbers from 1 to 20. Use a `for` loop to print each number.

4-7. Threes: Make a list of the multiples of 3, from 3 to 30. Use a `for` loop to print the numbers in your list.

4-8. Cubes: A number raised to the third power is called a *cube*. For example, the cube of 2 is written as `2**3` in Python. Make a list of the first 10 cubes (that is, the cube of each integer from 1 through 10), and use a `for` loop to print out the value of each cube.

4-9. Cube Comprehension: Use a list comprehension to generate a list of the first 10 cubes.

Working with Part of a List

In Chapter 3 you learned how to access single elements in a list, and in this chapter you've been learning how to work through all the elements in a list. You can also work with a specific group of items in a list, called a *slice* in Python.

Slicing a List

To make a slice, you specify the index of the first and last elements you want to work with. As with the `range()` function, Python stops one item before the second index you specify. To output the first three elements in a list, you would request indices `0` through `3`, which would return elements `0`, `1`, and `2`.

The following example involves a list of players on a team:

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[0:3])
```

This code prints a slice of the list. The output retains the structure of the list, and includes the first three players in the list:

```
['charles', 'martina', 'michael']
```

You can generate any subset of a list. For example, if you want the second, third, and fourth items in a list, you would start the slice at index `1` and end it at index `4`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

This time the slice starts with `'martina'` and ends with `'florence'`:

```
['martina', 'michael', 'florence']
```

If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Without a starting index, Python starts at the beginning of the list:

```
['charles', 'martina', 'michael', 'florence']
```

A similar syntax works if you want a slice that includes the end of a list. For example, if you want all items from the third item through the last item, you can start with index `2` and omit the second index:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python returns all items from the third item through the end of the list:

```
['michael', 'florence', 'eli']
```

This syntax allows you to output all of the elements from any point in your list to the end, regardless of the length of the list. Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

This prints the names of the last three players and will continue to work as the list of players changes in size.

NOTE *You can include a third value in the brackets indicating a slice. If a third value is included, this tells Python how many items to skip between items in the specified range.*

Looping Through a Slice

You can use a slice in a `for` loop if you want to loop through a subset of the elements in a list. In the next example, we loop through the first three players and print their names as part of a simple roster:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

Instead of looping through the entire list of players, Python loops through only the first three names ❶:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Slices are very useful in a number of situations. For instance, when you're creating a game, you could add a player's final score to a list every time that player finishes playing. You could then get a player's top three scores by sorting the list in decreasing order and taking a slice that includes just the first three scores. When you're working with data, you can use slices to process your data in chunks of a specific size. Or, when you're building a web application, you could use slices to display information in a series of pages with an appropriate amount of information on each page.

Copying a List

Often, you'll want to start with an existing list and make an entirely new list based on the first one. Let's explore how copying a list works and examine one situation in which copying a list is useful.

To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index ([:]). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list.

For example, imagine we have a list of our favorite foods and want to make a separate list of foods that a friend likes. This friend likes everything in our list so far, so we can create their list by copying ours:

foods.py

```
my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

First, we make a list of the foods we like called `my_foods`. Then we make a new list called `friend_foods`. We make a copy of `my_foods` by asking for a slice of `my_foods` without specifying any indices ❶, and assign the copy to `friend_foods`. When we print each list, we see that they both contain the same foods:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favorite foods:

```
my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli')
❸ friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

We copy the original items in `my_foods` to the new list `friend_foods`, as we did in the previous example ❶. Next, we add a new food to each list: we add `'cannoli'` to `my_foods` ❷, and we add `'ice cream'` to `friend_foods` ❸. We then print the two lists to see whether each of these foods is in the appropriate list:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

The output shows that `'cannoli'` now appears in our list of favorite foods but `'ice cream'` does not. We can see that `'ice cream'` now appears in our friend's list but `'cannoli'` does not. If we had simply set `friend_foods` equal to `my_foods`, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice:

```
my_foods = ['pizza', 'falafel', 'carrot cake']

# This doesn't work:
friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Instead of assigning a copy of `my_foods` to `friend_foods`, we set `friend_foods` equal to `my_foods`. This syntax actually tells Python to associate the new variable `friend_foods` with the list that is already associated with `my_foods`, so now both variables point to the same list. As a result, when we add `'cannoli'` to `my_foods`, it will also appear in `friend_foods`. Likewise `'ice cream'` will appear in both lists, even though it appears to be added only to `friend_foods`.

The output shows that both lists are the same now, which is not what we wanted:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

NOTE *Don't worry about the details in this example for now. If you're trying to work with a copy of a list and you see unexpected behavior, make sure you are copying the list using a slice, as we did in the first example.*

TRY IT YOURSELF

4-10. Slices: Using one of the programs you wrote in this chapter, add several lines to the end of the program that do the following:

- Print the message *The first three items in the list are:*. Then use a slice to print the first three items from that program's list.
- Print the message *Three items from the middle of the list are:*. Then use a slice to print three items from the middle of the list.
- Print the message *The last three items in the list are:*. Then use a slice to print the last three items in the list.

4-11. My Pizzas, Your Pizzas: Start with your program from Exercise 4-1 (page 56). Make a copy of the list of pizzas, and call it `friend_pizzas`. Then, do the following:

- Add a new pizza to the original list.
- Add a different pizza to the list `friend_pizzas`.
- Prove that you have two separate lists. Print the message *My favorite pizzas are:*, and then use a `for` loop to print the first list. Print the message *My friend's favorite pizzas are:*, and then use a `for` loop to print the second list. Make sure each new pizza is stored in the appropriate list.

4-12. More Loops: All versions of *foods.py* in this section have avoided using `for` loops when printing, to save space. Choose a version of *foods.py*, and write two `for` loops to print each list of foods.

Tuples

Lists work well for storing collections of items that can change throughout the life of a program. The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as *immutable*, and an immutable list is called a *tuple*.

Defining a Tuple

A tuple looks just like a list, except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

dimensions.py

```
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
```

We define the tuple `dimensions`, using parentheses instead of square brackets. Then we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
200
50
```

Let's see what happens if we try to change one of the items in the tuple `dimensions`:

```
dimensions = (200, 50)
dimensions[0] = 250
```

This code tries to change the value of the first dimension, but Python returns a type error. Because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

NOTE *Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple with one element, you need to include a trailing comma:*

```
my_t = (3,)
```

It doesn't often make sense to build a tuple with one element, but this can happen when tuples are generated automatically.

Looping Through All Values in a Tuple

You can loop over all the values in a tuple using a `for` loop, just as you did with a list:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
200
50
```

Writing Over a Tuple

Although you can't modify a tuple, you can assign a new value to a variable that represents a tuple. For example, if we wanted to change the dimensions of this rectangle, we could redefine the entire tuple:

```
dimensions = (200, 50)
print("Original dimensions:")
for dimension in dimensions:
    print(dimension)

dimensions = (400, 100)
print("\nModified dimensions:")
for dimension in dimensions:
    print(dimension)
```

The first four lines define the original tuple and print the initial dimensions. We then associate a new tuple with the variable `dimensions`, and print the new values. Python doesn't raise any errors this time, because reassigning a variable is valid:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.

TRY IT YOURSELF

4-13. Buffet: A buffet-style restaurant offers only five basic foods. Think of five simple foods, and store them in a tuple.

- Use a `for` loop to print each food the restaurant offers.
- Try to modify one of the items, and make sure that Python rejects the change.
- The restaurant changes its menu, replacing two of the items with different foods. Add a line that rewrites the tuple, and then use a `for` loop to print each of the items on the revised menu.

Styling Your Code

Now that you're writing longer programs, it's a good idea to learn how to style your code consistently. Take the time to make your code as easy as possible to read. Writing easy-to-read code helps you keep track of what your programs are doing and helps others understand your code as well.

Python programmers have agreed on a number of styling conventions to ensure that everyone's code is structured in roughly the same way. Once you've learned to write clean Python code, you should be able to understand the overall structure of anyone else's Python code, as long as they follow the same guidelines. If you're hoping to become a professional programmer at some point, you should begin following these guidelines as soon as possible to develop good habits.

The Style Guide

When someone wants to make a change to the Python language, they write a *Python Enhancement Proposal (PEP)*. One of the oldest PEPs is *PEP 8*, which instructs Python programmers on how to style their code. PEP 8 is fairly lengthy, but much of it relates to more complex coding structures than what you've seen so far.

The Python style guide was written with the understanding that code is read more often than it is written. You'll write your code once and then start reading it as you begin debugging. When you add features to a program, you'll spend more time reading your code. When you share your code with other programmers, they'll read your code as well.

Given the choice between writing code that's easier to write or code that's easier to read, Python programmers will almost always encourage you to write code that's easier to read. The following guidelines will help you write clear code from the start.

Indentation

PEP 8 recommends that you use four spaces per indentation level. Using four spaces improves readability while leaving room for multiple levels of indentation on each line.

In a word processing document, people often use tabs rather than spaces to indent. This works well for word processing documents, but the Python interpreter gets confused when tabs are mixed with spaces. Every text editor provides a setting that lets you use the TAB key but then converts each tab to a set number of spaces. You should definitely use your TAB key, but also make sure your editor is set to insert spaces rather than tabs into your document.

Mixing tabs and spaces in your file can cause problems that are very difficult to diagnose. If you think you have a mix of tabs and spaces, you can convert all tabs in a file to spaces in most editors.

Line Length

Many Python programmers recommend that each line should be less than 80 characters. Historically, this guideline developed because most computers could fit only 79 characters on a single line in a terminal window. Currently, people can fit much longer lines on their screens, but other reasons exist to adhere to the 79-character standard line length.

Professional programmers often have several files open on the same screen, and using the standard line length allows them to see entire lines in two or three files that are open side by side onscreen. PEP 8 also recommends that you limit all of your comments to 72 characters per line, because some of the tools that generate automatic documentation for larger projects add formatting characters at the beginning of each commented line.

The PEP 8 guidelines for line length are not set in stone, and some teams prefer a 99-character limit. Don't worry too much about line length in your code as you're learning, but be aware that people who are working collaboratively almost always follow the PEP 8 guidelines. Most editors allow you to set up a visual cue, usually a vertical line on your screen, that shows you where these limits are.

NOTE *Appendix B shows you how to configure your text editor so it always inserts four spaces each time you press the TAB key and shows a vertical guideline to help you follow the 79-character limit.*

Blank Lines

To group parts of your program visually, use blank lines. You should use blank lines to organize your files, but don't do so excessively. By following the examples provided in this book, you should strike the right balance. For example, if you have five lines of code that build a list and then another three lines that do something with that list, it's appropriate to place a blank line between the two sections. However, you should not place three or four blank lines between the two sections.

Blank lines won't affect how your code runs, but they will affect the readability of your code. The Python interpreter uses horizontal indentation to interpret the meaning of your code, but it disregards vertical spacing.

Other Style Guidelines

PEP 8 has many additional styling recommendations, but most of the guidelines refer to more complex programs than what you're writing at this point. As you learn more complex Python structures, I'll share the relevant parts of the PEP 8 guidelines.

TRY IT YOURSELF

4-14. PEP 8: Look through the original PEP 8 style guide at *https://python.org/dev/peps/pep-0008*. You won't use much of it now, but it might be interesting to skim through it.

4-15. Code Review: Choose three of the programs you've written in this chapter and modify each one to comply with PEP 8.

- Use four spaces for each indentation level. Set your text editor to insert four spaces every time you press the TAB key, if you haven't already done so (see Appendix B for instructions on how to do this).
- Use less than 80 characters on each line, and set your editor to show a vertical guideline at the 80th character position.
- Don't use blank lines excessively in your program files.

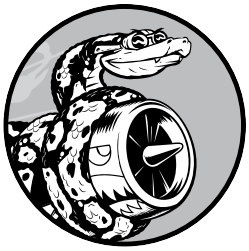
Summary

In this chapter, you learned how to work efficiently with the elements in a list. You learned how to work through a list using a `for` loop, how Python uses indentation to structure a program, and how to avoid some common indentation errors. You learned to make simple numerical lists, as well as a few operations you can perform on numerical lists. You learned how to slice a list to work with a subset of items and how to copy lists properly using a slice. You also learned about tuples, which provide a degree of protection to a set of values that shouldn't change, and how to style your increasingly complex code to make it easy to read.

In Chapter 5, you'll learn to respond appropriately to different conditions by using `if` statements. You'll learn to string together relatively complex sets of conditional tests to respond appropriately to exactly the kind of situation or information you're looking for. You'll also learn to use `if` statements while looping through a list to take specific actions with selected elements from a list.

5

IF STATEMENTS



Programming often involves examining a set of conditions and deciding which action to take based on those conditions. Python's `if` statement allows you to examine the current state of a program and respond appropriately to that state.

In this chapter, you'll learn to write conditional tests, which allow you to check any condition of interest. You'll learn to write simple `if` statements, and you'll learn how to create a more complex series of `if` statements to identify when the exact conditions you want are present. You'll then apply this concept to lists, so you'll be able to write a `for` loop that handles most items in a list one way but handles certain items with specific values in a different way.

A Simple Example

The following example shows how `if` tests let you respond to special situations correctly. Imagine you have a list of cars and you want to print out the name of each car. Car names are proper names, so the names of most cars should be printed in title case. However, the value `'bmw'` should be printed in all uppercase. The following code loops through a list of car names and looks for the value `'bmw'`. Whenever the value is `'bmw'`, it's printed in uppercase instead of title case:

cars.py

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

❶

The loop in this example first checks if the current value of `car` is `'bmw'` ❶. If it is, the value is printed in uppercase. If the value of `car` is anything other than `'bmw'`, it's printed in title case:

```
Audi
BMW
Subaru
Toyota
```

This example combines a number of the concepts you'll learn about in this chapter. Let's begin by looking at the kinds of tests you can use to examine the conditions in your program.

Conditional Tests

At the heart of every `if` statement is an expression that can be evaluated as `True` or `False` and is called a *conditional test*. Python uses the values `True` and `False` to decide whether the code in an `if` statement should be executed. If a conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

Checking for Equality

Most conditional tests compare the current value of a variable to a specific value of interest. The simplest conditional test checks whether the value of a variable is equal to the value of interest:

```
>>> car = 'bmw'
>>> car == 'bmw'
True
```

The first line sets the value of `car` to `'bmw'` using a single equal sign, as you've seen many times already. The next line checks whether the value of `car` is `'bmw'` by using a double equal sign (`==`). This *equality operator* returns `True` if the values on the left and right side of the operator match, and `False` if they don't match. The values in this example match, so Python returns `True`.

When the value of `car` is anything other than `'bmw'`, this test returns `False`:

```
>>> car = 'audi'
>>> car == 'bmw'
False
```

A single equal sign is really a statement; you might read the first line of code here as "Set the value of `car` equal to `'audi'`." On the other hand, a double equal sign asks a question: "Is the value of `car` equal to `'bmw'`?" Most programming languages use equal signs in this way.

Ignoring Case When Checking for Equality

Testing for equality is case sensitive in Python. For example, two values with different capitalization are not considered equal:

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

If case matters, this behavior is advantageous. But if case doesn't matter and instead you just want to test the value of a variable, you can convert the variable's value to lowercase before doing the comparison:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

This test will return `True` no matter how the value `'Audi'` is formatted because the test is now case insensitive. The `lower()` method doesn't change the value that was originally stored in `car`, so you can do this kind of comparison without affecting the original variable:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
>>> car
'Audi'
```

We first assign the capitalized string `'Audi'` to the variable `car`. Then, we convert the value of `car` to lowercase and compare the lowercase value to the string `'audi'`. The two strings match, so Python returns `True`. We can see that the value stored in `car` has not been affected by the `lower()` method.

Websites enforce certain rules for the data that users enter in a manner similar to this. For example, a site might use a conditional test like this to

ensure that every user has a truly unique username, not just a variation on the capitalization of another person's username. When someone submits a new username, that new username is converted to lowercase and compared to the lowercase versions of all existing usernames. During this check, a username like `'John'` will be rejected if any variation of `'john'` is already in use.

Checking for Inequality

When you want to determine whether two values are not equal, you can use the *inequality operator* (`!=`). Let's use another `if` statement to examine how to use the inequality operator. We'll store a requested pizza topping in a variable and then print a message if the person did not order anchovies:

toppings.py

```
requested_topping = 'mushrooms'

if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

This code compares the value of `requested_topping` to the value `'anchovies'`. If these two values do not match, Python returns `True` and executes the code following the `if` statement. If the two values match, Python returns `False` and does not run the code following the `if` statement.

Because the value of `requested_topping` is not `'anchovies'`, the `print()` function is executed:

```
Hold the anchovies!
```

Most of the conditional expressions you write will test for equality, but sometimes you'll find it more efficient to test for inequality.

Numerical Comparisons

Testing numerical values is pretty straightforward. For example, the following code checks whether a person is 18 years old:

```
>>> age = 18
>>> age == 18
True
```

You can also test to see if two numbers are not equal. For example, the following code prints a message if the given answer is not correct:

magic_number.py

```
answer = 17
if answer != 42:
    print("That is not the correct answer. Please try again!")
```

The conditional test passes, because the value of `answer` (`17`) is not equal to `42`. Because the test passes, the indented code block is executed:

```
That is not the correct answer. Please try again!
```

You can include various mathematical comparisons in your conditional statements as well, such as less than, less than or equal to, greater than, and greater than or equal to:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Each mathematical comparison can be used as part of an `if` statement, which can help you detect the exact conditions of interest.

Checking Multiple Conditions

You may want to check multiple conditions at the same time. For example, sometimes you might need two conditions to be `True` to take an action. Other times, you might be satisfied with just one condition being `True`. The keywords `and` and `or` can help you in these situations.

Using and to Check Multiple Conditions

To check whether two conditions are both `True` simultaneously, use the keyword `and` to combine the two conditional tests; if each test passes, the overall expression evaluates to `True`. If either test fails or if both tests fail, the expression evaluates to `False`.

For example, you can check whether two people are both over 21 by using the following test:

```
>>> age_0 = 22
>>> age_1 = 18
❶ >>> age_0 >= 21 and age_1 >= 21
False
❷ >>> age_1 = 22
>>> age_0 >= 21 and age_1 >= 21
True
```

First, we define two ages, `age_0` and `age_1`. Then we check whether both ages are 21 or older ❶. The test on the left passes, but the test on the right fails, so the overall conditional expression evaluates to `False`. We then change `age_1` to 22 ❷. The value of `age_1` is now greater than 21, so both individual tests pass, causing the overall conditional expression to evaluate as `True`.

To improve readability, you can use parentheses around the individual tests, but they are not required. If you use parentheses, your test would look like this:

```
(age_0 >= 21) and (age_1 >= 21)
```

Using or to Check Multiple Conditions

The keyword or allows you to check multiple conditions as well, but it passes when either or both of the individual tests pass. An or expression fails only when both individual tests fail.

Let's consider two ages again, but this time we'll look for only one person to be over 21:

```
>>> age_0 = 22
>>> age_1 = 18
❶ >>> age_0 >= 21 or age_1 >= 21
True
❷ >>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

We start with two age variables again. Because the test for age_0 ❶ passes, the overall expression evaluates to True. We then lower age_0 to 18. In the final test ❷, both tests now fail and the overall expression evaluates to False.

Checking Whether a Value Is in a List

Sometimes it's important to check whether a list contains a certain value before taking an action. For example, you might want to check whether a new username already exists in a list of current usernames before completing someone's registration on a website. In a mapping project, you might want to check whether a submitted location already exists in a list of known locations.

To find out whether a particular value is already in a list, use the keyword in. Let's consider some code you might write for a pizzeria. We'll make a list of toppings a customer has requested for a pizza and then check whether certain toppings are in the list.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
>>> 'mushrooms' in requested_toppings
True
>>> 'pepperoni' in requested_toppings
False
```

The keyword in tells Python to check for the existence of 'mushrooms' and 'pepperoni' in the list requested_toppings. This technique is quite powerful because you can create a list of essential values, and then easily check whether the value you're testing matches one of the values in the list.

Checking Whether a Value Is Not in a List

Other times, it's important to know if a value does not appear in a list. You can use the keyword not in this situation. For example, consider a list of users who are banned from commenting in a forum. You can check whether a user has been banned before allowing that person to submit a comment:

banned_users.py

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'
```

```
if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")
```

The `if` statement here reads quite clearly. If the value of `user` is not in the list `banned_users`, Python returns `True` and executes the indented line.

The user `'marie'` is not in the list `banned_users`, so she sees a message inviting her to post a response:

```
Marie, you can post a response if you wish.
```

Boolean Expressions

As you learn more about programming, you'll hear the term *Boolean expression* at some point. A Boolean expression is just another name for a conditional test. A *Boolean value* is either `True` or `False`, just like the value of a conditional expression after it has been evaluated.

Boolean values are often used to keep track of certain conditions, such as whether a game is running or whether a user can edit certain content on a website:

```
game_active = True
can_edit = False
```

Boolean values provide an efficient way to track the state of a program or a particular condition that is important in your program.

TRY IT YOURSELF

5-1. Conditional Tests: Write a series of conditional tests. Print a statement describing each test and your prediction for the results of each test. Your code should look something like this:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

- Look closely at your results, and make sure you understand why each line evaluates to `True` or `False`.
- Create at least 10 tests. Have at least 5 tests evaluate to `True` and another 5 tests evaluate to `False`.

(continued)

5-2. More Conditional Tests: You don't have to limit the number of tests you create to 10. If you want to try more comparisons, write more tests and add them to *conditional_tests.py*. Have at least one `True` and one `False` result for each of the following:

- Tests for equality and inequality with strings
- Tests using the `lower()` method
- Numerical tests involving equality and inequality, greater than and less than, greater than or equal to, and less than or equal to
- Tests using the `and` keyword and the `or` keyword
- Test whether an item is in a list
- Test whether an item is not in a list

if Statements

When you understand conditional tests, you can start writing `if` statements. Several different kinds of `if` statements exist, and your choice of which to use depends on the number of conditions you need to test. You saw several examples of `if` statements in the discussion about conditional tests, but now let's dig deeper into the topic.

Simple if Statements

The simplest kind of `if` statement has one test and one action:

```
if conditional_test:
    do something
```

You can put any conditional test in the first line and just about any action in the indented block following the test. If the conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

Let's say we have a variable representing a person's age, and we want to know if that person is old enough to vote. The following code tests whether the person can vote:

voting.py

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
```

Python checks to see whether the value of `age` is greater than or equal to 18. It is, so Python executes the indented `print()` call:

```
You are old enough to vote!
```

Indentation plays the same role in `if` statements as it did in `for` loops. All indented lines after an `if` statement will be executed if the test passes, and the entire block of indented lines will be ignored if the test does not pass.

You can have as many lines of code as you want in the block following the `if` statement. Let's add another line of output if the person is old enough to vote, asking if the individual has registered to vote yet:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

The conditional test passes, and both `print()` calls are indented, so both lines are printed:

```
You are old enough to vote!
Have you registered to vote yet?
```

If the value of `age` is less than 18, this program would produce no output.

if-else Statements

Often, you'll want to take one action when a conditional test passes and a different action in all other cases. Python's `if-else` syntax makes this possible. An `if-else` block is similar to a simple `if` statement, but the `else` statement allows you to define an action or set of actions that are executed when the conditional test fails.

We'll display the same message we had previously if the person is old enough to vote, but this time we'll add a message for anyone who is not old enough to vote:

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

If the conditional test ❶ passes, the first block of indented `print()` calls is executed. If the test evaluates to `False`, the `else` block ❷ is executed. Because `age` is less than 18 this time, the conditional test fails and the code in the `else` block is executed:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

This code works because it has only two possible situations to evaluate: a person is either old enough to vote or not old enough to vote. The `if-else`

structure works well in situations in which you want Python to always execute one of two possible actions. In a simple `if-else` chain like this, one of the two actions will always be executed.

The if-elif-else Chain

Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `if-elif-else` syntax. Python executes only one block in an `if-elif-else` chain. It runs each conditional test in order, until one passes. When a test passes, the code following that test is executed and Python skips the rest of the tests.

Many real-world situations involve more than two possible conditions. For example, consider an amusement park that charges different rates for different age groups:

- Admission for anyone under age 4 is free.
- Admission for anyone between the ages of 4 and 18 is $25.
- Admission for anyone age 18 or older is $40.

How can we use an `if` statement to determine a person's admission rate? The following code tests for the age group of a person and then prints an admission price message:

amusement_park.py

```
age = 12
❶ if age < 4:
    print("Your admission cost is $0.")
❷ elif age < 18:
    print("Your admission cost is $25.")
❸ else:
    print("Your admission cost is $40.")
```

The `if` test ❶ checks whether a person is under 4 years old. When the test passes, an appropriate message is printed and Python skips the rest of the tests. The `elif` line ❷ is really another `if` test, which runs only if the previous test failed. At this point in the chain, we know the person is at least 4 years old because the first test failed. If the person is under 18, an appropriate message is printed and Python skips the `else` block. If both the `if` and `elif` tests fail, Python runs the code in the `else` block ❸.

In this example the `if` test ❶ evaluates to `False`, so its code block is not executed. However, the `elif` test evaluates to `True` (12 is less than 18) so its code is executed. The output is one sentence, informing the user of the admission cost:

```
Your admission cost is $25.
```

Any age greater than 17 would cause the first two tests to fail. In these situations, the `else` block would be executed and the admission price would be $40.

Rather than printing the admission price within the `if-elif-else` block, it would be more concise to set just the price inside the `if-elif-else` chain

and then have a single `print()` call that runs after the chain has been evaluated:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40

print(f"Your admission cost is ${price}.")
```

The indented lines set the value of `price` according to the person's age, as in the previous example. After the price is set by the `if-elif-else` chain, a separate unindented `print()` call uses this value to display a message reporting the person's admission price.

This code produces the same output as the previous example, but the purpose of the `if-elif-else` chain is narrower. Instead of determining a price and displaying a message, it simply determines the admission price. In addition to being more efficient, this revised code is easier to modify than the original approach. To change the text of the output message, you would need to change only one `print()` call rather than three separate `print()` calls.

Using Multiple elif Blocks

You can use as many `elif` blocks in your code as you like. For example, if the amusement park were to implement a discount for seniors, you could add one more conditional test to the code to determine whether someone qualifies for the senior discount. Let's say that anyone 65 or older pays half the regular admission, or $20:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
else:
    price = 20

print(f"Your admission cost is ${price}.")
```

Most of this code is unchanged. The second `elif` block now checks to make sure a person is less than age 65 before assigning them the full admission rate of $40. Notice that the value assigned in the `else` block needs to be changed to $20, because the only ages that make it to this block are for people 65 or older.

Omitting the else Block

Python does not require an `else` block at the end of an `if-elif` chain. Sometimes, an `else` block is useful. Other times, it's clearer to use an additional `elif` statement that catches the specific condition of interest:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f"Your admission cost is ${price}.")
```

The final `elif` block assigns a price of $20 when the person is 65 or older, which is a little clearer than the general `else` block. With this change, every block of code must pass a specific test in order to be executed.

The `else` block is a catchall statement. It matches any condition that wasn't matched by a specific `if` or `elif` test, and that can sometimes include invalid or even malicious data. If you have a specific final condition you're testing for, consider using a final `elif` block and omit the `else` block. As a result, you'll be more confident that your code will run only under the correct conditions.

Testing Multiple Conditions

The `if-elif-else` chain is powerful, but it's only appropriate to use when you just need one test to pass. As soon as Python finds one test that passes, it skips the rest of the tests. This behavior is beneficial, because it's efficient and allows you to test for one specific condition.

However, sometimes it's important to check all conditions of interest. In this case, you should use a series of simple `if` statements with no `elif` or `else` blocks. This technique makes sense when more than one condition could be `True`, and you want to act on every condition that is `True`.

Let's reconsider the pizzeria example. If someone requests a two-topping pizza, you'll need to be sure to include both toppings on their pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
❶ if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
```

```
if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

We start with a list containing the requested toppings. The first `if` statement checks to see whether the person requested mushrooms on their pizza. If so, a message is printed confirming that topping. The test for pepperoni ❶ is another simple `if` statement, not an `elif` or `else` statement, so this test is run regardless of whether the previous test passed or not. The last `if` statement checks whether extra cheese was requested, regardless of the results from the first two tests. These three independent tests are executed every time this program is run.

Because every condition in this example is evaluated, both mushrooms and extra cheese are added to the pizza:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

This code would not work properly if we used an `if-elif-else` block, because the code would stop running after only one test passes. Here's what that would look like:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

The test for `'mushrooms'` is the first test to pass, so mushrooms are added to the pizza. However, the values `'extra cheese'` and `'pepperoni'` are never checked, because Python doesn't run any tests beyond the first test that passes in an `if-elif-else` chain. The customer's first topping will be added, but all of their other toppings will be missed:

```
Adding mushrooms.

Finished making your pizza!
```

In summary, if you want only one block of code to run, use an `if-elif-else` chain. If more than one block of code needs to run, use a series of independent `if` statements.

TRY IT YOURSELF

5-3. Alien Colors #1: Imagine an alien was just shot down in a game. Create a variable called `alien_color` and assign it a value of `'green'`, `'yellow'`, or `'red'`.

- Write an `if` statement to test whether the alien's color is green. If it is, print a message that the player just earned 5 points.
- Write one version of this program that passes the `if` test and another that fails. (The version that fails will have no output.)

5-4. Alien Colors #2: Choose a color for an alien as you did in Exercise 5-3, and write an `if-else` chain.

- If the alien's color is green, print a statement that the player just earned 5 points for shooting the alien.
- If the alien's color isn't green, print a statement that the player just earned 10 points.
- Write one version of this program that runs the `if` block and another that runs the `else` block.

5-5. Alien Colors #3: Turn your `if-else` chain from Exercise 5-4 into an `if-elif-else` chain.

- If the alien is green, print a message that the player earned 5 points.
- If the alien is yellow, print a message that the player earned 10 points.
- If the alien is red, print a message that the player earned 15 points.
- Write three versions of this program, making sure each message is printed for the appropriate color alien.

5-6. Stages of Life: Write an `if-elif-else` chain that determines a person's stage of life. Set a value for the variable `age`, and then:

- If the person is less than 2 years old, print a message that the person is a baby.
- If the person is at least 2 years old but less than 4, print a message that the person is a toddler.
- If the person is at least 4 years old but less than 13, print a message that the person is a kid.
- If the person is at least 13 years old but less than 20, print a message that the person is a teenager.
- If the person is at least 20 years old but less than 65, print a message that the person is an adult.
- If the person is age 65 or older, print a message that the person is an elder.

5-7. Favorite Fruit: Make a list of your favorite fruits, and then write a series of independent `if` statements that check for certain fruits in your list.

- Make a list of your three favorite fruits and call it `favorite_fruits`.
- Write five `if` statements. Each should check whether a certain kind of fruit is in your list. If the fruit is in your list, the `if` block should print a statement, such as *You really like bananas!*

Using if Statements with Lists

You can do some interesting work when you combine lists and `if` statements. You can watch for special values that need to be treated differently than other values in the list. You can efficiently manage changing conditions, such as the availability of certain items in a restaurant throughout a shift. You can also begin to prove that your code works as you expect it to in all possible situations.

Checking for Special Items

This chapter began with a simple example that showed how to handle a special value like `'bmw'`, which needed to be printed in a different format than other values in the list. Now that you have a basic understanding of conditional tests and `if` statements, let's take a closer look at how you can watch for special values in a list and handle those values appropriately.

Let's continue with the pizzeria example. The pizzeria displays a message whenever a topping is added to your pizza, as it's being made. The code for this action can be written very efficiently by making a list of toppings the customer has requested and using a loop to announce each topping as it's added to the pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

The output is straightforward because this code is just a simple `for` loop:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

But what if the pizzeria runs out of green peppers? An `if` statement inside the `for` loop can handle this situation appropriately:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

This time, we check each requested item before adding it to the pizza. The `if` statement checks to see if the person requested green peppers. If so, we display a message informing them why they can't have green peppers. The `else` block ensures that all other toppings will be added to the pizza.

The output shows that each requested topping is handled appropriately.

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Checking That a List Is Not Empty

We've made a simple assumption about every list we've worked with so far: we've assumed that each list has at least one item in it. Soon we'll let users provide the information that's stored in a list, so we won't be able to assume that a list has any items in it each time a loop is run. In this situation, it's useful to check whether a list is empty before running a `for` loop.

As an example, let's check whether the list of requested toppings is empty before building the pizza. If the list is empty, we'll prompt the user and make sure they want a plain pizza. If the list is not empty, we'll build the pizza just as we did in the previous examples:

```
requested_toppings = []

if requested_toppings:
    for requested_topping in requested_toppings:
        print(f"Adding {requested_topping}.")
    print("\nFinished making your pizza!")
else:
    print("Are you sure you want a plain pizza?")
```

This time we start out with an empty list of requested toppings. Instead of jumping right into a `for` loop, we do a quick check first. When the name of a list is used in an `if` statement, Python returns `True` if the list contains at least one item; an empty list evaluates to `False`. If `requested_toppings` passes the conditional test, we run the same `for` loop we used in the previous

example. If the conditional test fails, we print a message asking the customer if they really want a plain pizza with no toppings.

The list is empty in this case, so the output asks if the user really wants a plain pizza:

```
Are you sure you want a plain pizza?
```

If the list is not empty, the output will show each requested topping being added to the pizza.

Using Multiple Lists

People will ask for just about anything, especially when it comes to pizza toppings. What if a customer actually wants french fries on their pizza? You can use lists and `if` statements to make sure your input makes sense before you act on it.

Let's watch out for unusual topping requests before we build a pizza. The following example defines two lists. The first is a list of available toppings at the pizzeria, and the second is the list of toppings that the user has requested. This time, each item in `requested_toppings` is checked against the list of available toppings before it's added to the pizza:

```
available_toppings = ['mushrooms', 'olives', 'green peppers',
                      'pepperoni', 'pineapple', 'extra cheese']

❶ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

for requested_topping in requested_toppings:
❷     if requested_topping in available_toppings:
          print(f"Adding {requested_topping}.")
❸     else:
          print(f"Sorry, we don't have {requested_topping}.")

print("\nFinished making your pizza!")
```

First, we define a list of available toppings at this pizzeria. Note that this could be a tuple if the pizzeria has a stable selection of toppings. Then, we make a list of toppings that a customer has requested. There's an unusual request for a topping in this example: `'french fries'` ❶. Next we loop through the list of requested toppings. Inside the loop, we check to see if each requested topping is actually in the list of available toppings ❷. If it is, we add that topping to the pizza. If the requested topping is not in the list of available toppings, the `else` block will run ❸. The `else` block prints a message telling the user which toppings are unavailable.

This code syntax produces clean, informative output:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

In just a few lines of code, we've managed a real-world situation pretty effectively!

TRY IT YOURSELF

5-8. Hello Admin: Make a list of five or more usernames, including the name `'admin'`. Imagine you are writing code that will print a greeting to each user after they log in to a website. Loop through the list, and print a greeting to each user.

- If the username is `'admin'`, print a special greeting, such as *Hello admin, would you like to see a status report?*
- Otherwise, print a generic greeting, such as *Hello Jaden, thank you for logging in again.*

5-9. No Users: Add an `if` test to *hello_admin.py* to make sure the list of users is not empty.

- If the list is empty, print the message *We need to find some users!*
- Remove all of the usernames from your list, and make sure the correct message is printed.

5-10. Checking Usernames: Do the following to create a program that simulates how websites ensure that everyone has a unique username.

- Make a list of five or more usernames called `current_users`.
- Make another list of five usernames called `new_users`. Make sure one or two of the new usernames are also in the `current_users` list.
- Loop through the `new_users` list to see if each new username has already been used. If it has, print a message that the person will need to enter a new username. If a username has not been used, print a message saying that the username is available.
- Make sure your comparison is case insensitive. If `'John'` has been used, `'JOHN'` should not be accepted. (To do this, you'll need to make a copy of `current_users` containing the lowercase versions of all existing users.)

5-11. Ordinal Numbers: Ordinal numbers indicate their position in a list, such as *1st* or *2nd*. Most ordinal numbers end in *th*, except 1, 2, and 3.

- Store the numbers 1 through 9 in a list.
- Loop through the list.
- Use an `if-elif-else` chain inside the loop to print the proper ordinal ending for each number. Your output should read `"1st 2nd 3rd 4th 5th 6th 7th 8th 9th"`, and each result should be on a separate line.

Styling Your if Statements

In every example in this chapter, you've seen good styling habits. The only recommendation PEP 8 provides for styling conditional tests is to use a single space around comparison operators, such as ==, >=, and <=. For example:

```
if age < 4:
```

is better than:

```
if age<4:
```

Such spacing does not affect the way Python interprets your code; it just makes your code easier for you and others to read.

TRY IT YOURSELF

5-12. Styling `if` Statements: Review the programs you wrote in this chapter, and make sure you styled your conditional tests appropriately.

5-13. Your Ideas: At this point, you're a more capable programmer than you were when you started this book. Now that you have a better sense of how real-world situations are modeled in programs, you might be thinking of some problems you could solve with your own programs. Record any new ideas you have about problems you might want to solve as your programming skills continue to improve. Consider games you might want to write, datasets you might want to explore, and web applications you'd like to create.

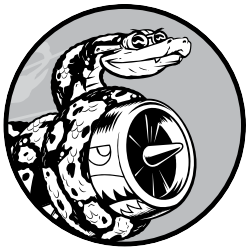
Summary

In this chapter you learned how to write conditional tests, which always evaluate to `True` or `False`. You learned to write simple `if` statements, `if-else` chains, and `if-elif-else` chains. You began using these structures to identify particular conditions you need to test and to know when those conditions have been met in your programs. You learned to handle certain items in a list differently than all other items while continuing to utilize the efficiency of a `for` loop. You also revisited Python's style recommendations to ensure that your increasingly complex programs are still relatively easy to read and understand.

In Chapter 6 you'll learn about Python's dictionaries. A dictionary is similar to a list, but it allows you to connect pieces of information. You'll learn how to build dictionaries, loop through them, and use them in combination with lists and `if` statements. Learning about dictionaries will enable you to model an even wider variety of real-world situations.

6

DICTIONARIES



In this chapter you'll learn how to use Python's dictionaries, which allow you to connect pieces of related information. You'll learn how to access the information once it's in a dictionary and how to modify that information. Because dictionaries can store an almost limitless amount of information, I'll show you how to loop through the data in a dictionary. Additionally, you'll learn to nest dictionaries inside lists, lists inside dictionaries, and even dictionaries inside other dictionaries.

Understanding dictionaries allows you to model a variety of real-world objects more accurately. You'll be able to create a dictionary representing a person and then store as much information as you want about that person. You can store their name, age, location, profession, and any other aspect of a person you can describe. You'll be able to store any two kinds of information that can be matched up, such as a list of words and their meanings, a list of people's names and their favorite numbers, a list of mountains and their elevations, and so forth.

A Simple Dictionary

Consider a game featuring aliens that can have different colors and point values. This simple dictionary stores information about a particular alien:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's color and point value. The last two lines access and display that information, as shown here:

```
green
5
```

As with most new programming concepts, using dictionaries takes practice. Once you've worked with dictionaries for a bit, you'll see how effectively they can model real-world situations.

Working with Dictionaries

A *dictionary* in Python is a collection of *key-value pairs*. Each *key* is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.

In Python, a dictionary is wrapped in braces ({}) with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'color': 'green', 'points': 5}
```

A *key-value pair* is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.

The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'color': 'green'}
```

This dictionary stores one piece of information about `alien_0`: the alien's color. The string `'color'` is a key in this dictionary, and its associated value is `'green'`.

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

alien.py

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

This returns the value associated with the key `'color'` from the dictionary `alien_0`:

```
green
```

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original `alien_0` dictionary with two key-value pairs:

```
alien_0 = {'color': 'green', 'points': 5}
```

Now you can access either the color or the point value of `alien_0`. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
alien_0 = {'color': 'green', 'points': 5}

new_points = alien_0['points']
print(f"You just earned {new_points} points!")
```

Once the dictionary has been defined, we pull the value associated with the key `'points'` from the dictionary. This value is then assigned to the variable `new_points`. The last line prints a statement about how many points the player just earned:

```
You just earned 5 points!
```

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. To add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets, along with the new value.

Let's add two new pieces of information to the `alien_0` dictionary: the alien's *x*- and *y*-coordinates, which will help us display the alien at a particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left edge of the screen by setting the *x*-coordinate to 0 and 25 pixels from the top by setting its *y*-coordinate to positive 25, as shown here:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

alien_0['x_position'] = 0
alien_0['y_position'] = 25
print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. Next, we add a new key-value pair to the dictionary: the key `'x_position'` and the value `0`. We do the same for the key `'y_position'`. When we print the modified dictionary, we see the two additional key-value pairs:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}
```

The final version of the dictionary contains four key-value pairs. The original two specify color and point value, and two more specify the alien's position.

Dictionaries retain the order in which they were defined. When you print a dictionary or loop through its elements, you will see the elements in the same order they were added to the dictionary.

Starting with an Empty Dictionary

It's sometimes convenient, or even necessary, to start with an empty dictionary and then add each new item to it. To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

alien.py

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

We first define an empty `alien_0` dictionary, and then add color and point values to it. The result is the dictionary we've been using in previous examples:

```
{'color': 'green', 'points': 5}
```

Typically, you'll use empty dictionaries when storing user-supplied data in a dictionary or when writing code that generates a large number of key-value pairs automatically.

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")
```

```
alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

We first define a dictionary for `alien_0` that contains only the alien's color; then we change the value associated with the key `'color'` to `'yellow'`. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

For a more interesting example, let's track the position of an alien that can move at different speeds. We'll store a value representing the alien's current speed and then use it to determine how far to the right the alien should move:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")

# Move the alien to the right.
# Determine how far to move the alien based on its current speed.
❶ if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # This must be a fast alien.
    x_increment = 3

# The new position is the old position plus the increment.
❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")
```

We start by defining an alien with an initial *x* position and *y* position, and a speed of `'medium'`. We've omitted the color and point values for the sake of simplicity, but this example would work the same way if you included those key-value pairs as well. We also print the original value of `x_position` to see how far the alien moves to the right.

An `if-elif-else` chain determines how far the alien should move to the right, and assigns this value to the variable `x_increment` ❶. If the alien's speed is `'slow'`, it moves one unit to the right; if the speed is `'medium'`, it moves two units to the right; and if it's `'fast'`, it moves three units to the right. Once the increment has been calculated, it's added to the value of `x_position` ❷, and the result is stored in the dictionary's `x_position`.

Because this is a medium-speed alien, its position shifts two units to the right:

```
Original x-position: 0
New x-position: 2
```

This technique is pretty cool: by changing one value in the alien's dictionary, you can change the overall behavior of the alien. For example, to turn this medium-speed alien into a fast alien, you would add this line:

```
alien_0['speed'] = 'fast'
```

The `if-elif-else` block would then assign a larger value to `x_increment` the next time the code runs.

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the `del` statement to completely remove a key-value pair. All `del` needs is the name of the dictionary and the key that you want to remove.

For example, let's remove the key `'points'` from the `alien_0` dictionary, along with its value:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

❶ del alien_0['points']
print(alien_0)
```

The `del` statement ❶ tells Python to delete the key `'points'` from the dictionary `alien_0` and to remove the value associated with that key as well. The output shows that the key `'points'` and its value of `5` are deleted from the dictionary, but the rest of the dictionary is unaffected:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

NOTE *Be aware that the deleted key-value pair is removed permanently.*

A Dictionary of Similar Objects

The previous example involved storing different kinds of information about one object, an alien in a game. You can also use a dictionary to store one kind of information about many objects. For example, say you want to poll a number of people and ask them what their favorite programming language is. A dictionary is useful for storing the results of a simple poll, like this:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }
```

As you can see, we've broken a larger dictionary into several lines. Each key is the name of a person who responded to the poll, and each value is their language choice. When you know you'll need more than one line to define a dictionary, press ENTER after the opening brace. Then indent the next line one level (four spaces) and write the first key-value pair, followed by a comma. From this point forward when you press ENTER, your text editor should automatically indent all subsequent key-value pairs to match the first key-value pair.

Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair, and indent it one level so it aligns with the keys in the dictionary. It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.

NOTE *Most editors have some functionality that helps you format extended lists and dictionaries in a similar manner to this example. Other acceptable ways to format long dictionaries are available as well, so you may see slightly different formatting in your editor, or in other sources.*

To use this dictionary, given the name of a person who took the poll, you can easily look up their favorite language:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

❶ language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}.")
```

To see which language Sarah chose, we ask for the value at:

```
favorite_languages['sarah']
```

We use this syntax to pull Sarah's favorite language from the dictionary ❶ and assign it to the variable `language`. Creating a new variable here makes for a much cleaner `print()` call. The output shows Sarah's favorite language:

```
Sarah's favorite language is C.
```

You could use this same syntax with any individual represented in the dictionary.

Using get() to Access Values

Using keys in square brackets to retrieve the value you're interested in from a dictionary might cause one potential problem: if the key you ask for doesn't exist, you'll get an error.

Let's see what happens when you ask for the point value of an alien that doesn't have a point value set:

alien_no_points.py

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

This results in a traceback, showing a `KeyError`:

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
          ~~~~~~~^^^^^^^^^^
KeyError: 'points'
```

You'll learn more about how to handle errors like this in general in Chapter 10. For dictionaries specifically, you can use the `get()` method to set a default value that will be returned if the requested key doesn't exist.

The `get()` method requires a key as a first argument. As a second optional argument, you can pass the value to be returned if the key doesn't exist:

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

If the key `'points'` exists in the dictionary, you'll get the corresponding value. If it doesn't, you get the default value. In this case, `points` doesn't exist, and we get a clean message instead of an error:

```
No point value assigned.
```

If there's a chance the key you're asking for might not exist, consider using the `get()` method instead of the square bracket notation.

NOTE *If you leave out the second argument in the call to `get()` and the key doesn't exist, Python will return the value `None`. The special value `None` means "no value exists." This is not an error: it's a special value meant to indicate the absence of a value. You'll see more uses for `None` in Chapter 8.*

TRY IT YOURSELF

6-1. Person: Use a dictionary to store information about a person you know. Store their first name, last name, age, and the city in which they live. You should have keys such as `first_name`, `last_name`, `age`, and `city`. Print each piece of information stored in your dictionary.

6-2. Favorite Numbers: Use a dictionary to store people's favorite numbers. Think of five names, and use them as keys in your dictionary. Think of a favorite

number for each person, and store each as a value in your dictionary. Print each person's name and their favorite number. For even more fun, poll a few friends and get some actual data for your program.

6-3. Glossary: A Python dictionary can be used to model an actual dictionary. However, to avoid confusion, let's call it a glossary.

- Think of five programming words you've learned about in the previous chapters. Use these words as the keys in your glossary, and store their meanings as values.
- Print each word and its meaning as neatly formatted output. You might print the word followed by a colon and then its meaning, or print the word on one line and then print its meaning indented on a second line. Use the newline character (\n) to insert a blank line between each word-meaning pair in your output.

Looping Through a Dictionary

A single Python dictionary can contain just a few key-value pairs or millions of pairs. Because a dictionary can contain large amounts of data, Python lets you loop through a dictionary. Dictionaries can be used to store information in a variety of ways; therefore, several different ways exist to loop through them. You can loop through all of a dictionary's key-value pairs, through its keys, or through its values.

Looping Through All Key-Value Pairs

Before we explore the different approaches to looping, let's consider a new dictionary designed to store information about a user on a website. The following dictionary would store one person's username, first name, and last name:

user.py

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

You can access any single piece of information about `user_0` based on what you've already learned in this chapter. But what if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a `for` loop:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

```
for key, value in user_0.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

To write a `for` loop for a dictionary, you create names for the two variables that will hold the key and value in each key-value pair. You can choose any names you want for these two variables. This code would work just as well if you had used abbreviations for the variable names, like this:

```
for k, v in user_0.items()
```

The second half of the `for` statement includes the name of the dictionary followed by the method `items()`, which returns a sequence of key-value pairs. The `for` loop then assigns each of these pairs to the two variables provided. In the preceding example, we use the variables to print each `key`, followed by the associated `value`. The `"\n"` in the first `print()` call ensures that a blank line is inserted before each key-value pair in the output:

```
Key: username
Value: efermi

Key: first
Value: enrico

Key: last
Value: fermi
```

Looping through all key-value pairs works particularly well for dictionaries like the *favorite_languages.py* example on page 96, which stores the same kind of information for many different keys. If you loop through the `favorite_languages` dictionary, you get the name of each person in the dictionary and their favorite programming language. Because the keys always refer to a person's name and the value is always a language, we'll use the variables `name` and `language` in the loop instead of `key` and `value`. This will make it easier to follow what's happening inside the loop:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

for name, language in favorite_languages.items():
    print(f"{name.title()}'s favorite language is {language.title()}.")
```

This code tells Python to loop through each key-value pair in the dictionary. As it works through each pair the key is assigned to the variable `name`, and the value is assigned to the variable `language`. These descriptive names make it much easier to see what the `print()` call is doing.

Now, in just a few lines of code, we can display all of the information from the poll:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Rust.
Phil's favorite language is Python.
```

This type of looping would work just as well if our dictionary stored the results from polling a thousand or even a million people.

Looping Through All the Keys in a Dictionary

The `keys()` method is useful when you don't need to work with all of the values in a dictionary. Let's loop through the `favorite_languages` dictionary and print the names of everyone who took the poll:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

for name in favorite_languages.keys():
    print(name.title())
```

This `for` loop tells Python to pull all the keys from the dictionary `favorite_languages` and assign them one at a time to the variable `name`. The output shows the names of everyone who took the poll:

```
Jen
Sarah
Edward
Phil
```

Looping through the keys is actually the default behavior when looping through a dictionary, so this code would have exactly the same output if you wrote:

```
for name in favorite_languages:
```

rather than:

```
for name in favorite_languages.keys():
```

You can choose to use the `keys()` method explicitly if it makes your code easier to read, or you can omit it if you wish.

You can access the value associated with any key you care about inside the loop, by using the current key. Let's print a message to a couple of friends about the languages they chose. We'll loop through the names in

the dictionary as we did previously, but when the name matches one of our friends, we'll display a message about their favorite language:

```
favorite_languages = {
    --snip--
    }

friends = ['phil', 'sarah']
for name in favorite_languages.keys():
    print(f"Hi {name.title()}.")

❶   if name in friends:
❷       language = favorite_languages[name].title()
        print(f"\t{name.title()}, I see you love {language}!")
```

First, we make a list of friends that we want to print a message to. Inside the loop, we print each person's name. Then we check whether the `name` we're working with is in the list `friends` ❶. If it is, we determine the person's favorite language using the name of the dictionary and the current value of `name` as the key ❷. We then print a special greeting, including a reference to their language of choice.

Everyone's name is printed, but our friends receive a special message:

```
Hi Jen.
Hi Sarah.
    Sarah, I see you love C!
Hi Edward.
Hi Phil.
    Phil, I see you love Python!
```

You can also use the `keys()` method to find out if a particular person was polled. This time, let's find out if Erin took the poll:

```
favorite_languages = {
    --snip--
    }

if 'erin' not in favorite_languages.keys():
    print("Erin, please take our poll!")
```

The `keys()` method isn't just for looping: it actually returns a sequence of all the keys, and the `if` statement simply checks if `'erin'` is in this sequence. Because she's not, a message is printed inviting her to take the poll:

```
Erin, please take our poll!
```

Looping Through a Dictionary's Keys in a Particular Order

Looping through a dictionary returns the items in the same order they were inserted. Sometimes, though, you'll want to loop through a dictionary in a different order.

One way to do this is to sort the keys as they're returned in the `for` loop. You can use the `sorted()` function to get a copy of the keys in order:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

for name in sorted(favorite_languages.keys()):
    print(f"{name.title()}, thank you for taking the poll.")
```

This `for` statement is like other `for` statements, except that we've wrapped the `sorted()` function around the `dictionary.keys()` method. This tells Python to get all the keys in the dictionary and sort them before starting the loop. The output shows everyone who took the poll, with the names displayed in order:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Looping Through All Values in a Dictionary

If you are primarily interested in the values that a dictionary contains, you can use the `values()` method to return a sequence of values without any keys. For example, say we simply want a list of all languages chosen in our programming language poll, without the name of the person who chose each language:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
for language in favorite_languages.values():
    print(language.title())
```

The `for` statement here pulls each value from the dictionary and assigns it to the variable `language`. When these values are printed, we get a list of all chosen languages:

```
The following languages have been mentioned:
Python
C
Rust
Python
```

This approach pulls all the values from the dictionary without checking for repeats. This might work fine with a small number of values, but in a poll with a large number of respondents, it would result in a very repetitive list. To see each language chosen without repetition, we can use a set. A *set* is a collection in which each item must be unique:

```
favorite_languages = {
    --snip--
    }

print("The following languages have been mentioned:")
for language in set(favorite_languages.values()):
    print(language.title())
```

When you wrap `set()` around a collection of values that contains duplicate items, Python identifies the unique items in the collection and builds a set from those items. Here we use `set()` to pull out the unique languages in `favorite_languages.values()`.

The result is a nonrepetitive list of languages that have been mentioned by people taking the poll:

```
The following languages have been mentioned:
Python
C
Rust
```

As you continue learning about Python, you'll often find a built-in feature of the language that helps you do exactly what you want with your data.

NOTE *You can build a set directly using braces and separating the elements with commas:*

```
>>> languages = {'python', 'rust', 'python', 'c'}
>>> languages
{'rust', 'python', 'c'}
```

It's easy to mistake sets for dictionaries because they're both wrapped in braces. When you see braces but no key-value pairs, you're probably looking at a set. Unlike lists and dictionaries, sets do not retain items in any specific order.

TRY IT YOURSELF

6-4. Glossary 2: Now that you know how to loop through a dictionary, clean up the code from Exercise 6-3 (page 99) by replacing your series of `print()` calls with a loop that runs through the dictionary's keys and values. When you're sure that your loop works, add five more Python terms to your glossary. When you run your program again, these new words and meanings should automatically be included in the output.

6-5. Rivers: Make a dictionary containing three major rivers and the country each river runs through. One key-value pair might be `'nile': 'egypt'`.

- Use a loop to print a sentence about each river, such as *The Nile runs through Egypt.*
- Use a loop to print the name of each river included in the dictionary.
- Use a loop to print the name of each country included in the dictionary.

6-6. Polling: Use the code in *favorite_languages.py* (page 96).

- Make a list of people who should take the favorite languages poll. Include some names that are already in the dictionary and some that are not.
- Loop through the list of people who should take the poll. If they have already taken the poll, print a message thanking them for responding. If they have not yet taken the poll, print a message inviting them to take the poll.

Nesting

Sometimes you'll want to store multiple dictionaries in a list, or a list of items as a value in a dictionary. This is called *nesting*. You can nest dictionaries inside a list, a list of items inside a dictionary, or even a dictionary inside another dictionary. Nesting is a powerful feature, as the following examples will demonstrate.

A List of Dictionaries

The `alien_0` dictionary contains a variety of information about one alien, but it has no room to store information about a second alien, much less a screen full of aliens. How can you manage a fleet of aliens? One way is to make a list of aliens in which each alien is a dictionary of information about that alien. For example, the following code builds a list of three aliens:

aliens.py

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

❶ aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)
```

We first create three dictionaries, each representing a different alien. We store each of these dictionaries in a list called `aliens` ❶. Finally, we loop through the list and print out each alien:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example, we use `range()` to create a fleet of 30 aliens:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
❶ for alien_number in range(30):
❷     new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
❸     aliens.append(new_alien)

# Show the first 5 aliens.
❹ for alien in aliens[:5]:
      print(alien)
print("...")

# Show how many aliens have been created.
print(f"Total number of aliens: {len(aliens)}")
```

This example begins with an empty list to hold all of the aliens that will be created. The `range()` function ❶ returns a series of numbers, which just tells Python how many times we want the loop to repeat. Each time the loop runs, we create a new alien ❷ and then append each new alien to the list `aliens` ❸. We use a slice to print the first five aliens ❹, and finally, we print the length of the list to prove we've actually generated the full fleet of 30 aliens:

```
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...

Total number of aliens: 30
```

These aliens all have the same characteristics, but Python considers each one a separate object, which allows us to modify each alien individually.

How might you work with a group of aliens like this? Imagine that one aspect of a game has some aliens changing color and moving faster as the game progresses. When it's time to change colors, we can use a `for` loop and an `if` statement to change the color of the aliens. For example, to change

the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range (30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Show the first 5 aliens.
for alien in aliens[:5]:
    print(alien)
print("...")
```

Because we want to modify the first three aliens, we loop through a slice that includes only the first three aliens. All of the aliens are green now, but that won't always be the case, so we write an `if` statement to make sure we're only modifying green aliens. If the alien is green, we change the color to `'yellow'`, the speed to `'medium'`, and the point value to `10`, as shown in the following output:

```
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...
```

You could expand this loop by adding an `elif` block that turns yellow aliens into red, fast-moving ones worth 15 points each. Without showing the entire program again, that loop would look like this:

```
for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10
    elif alien['color'] == 'yellow':
        alien['color'] = 'red'
        alien['speed'] = 'fast'
        alien['points'] = 15
```

It's common to store a number of dictionaries in a list when each dictionary contains many kinds of information about one object. For example, you might create a dictionary for each user on a website, as we did in *user.py*

on page 99, and store the individual dictionaries in a list called `users`. All of the dictionaries in the list should have an identical structure, so you can loop through the list and work with each dictionary object in the same way.

A List in a Dictionary

Rather than putting a dictionary inside a list, it's sometimes useful to put a list inside a dictionary. For example, consider how you might describe a pizza that someone is ordering. If you were to use only a list, all you could really store is a list of the pizza's toppings. With a dictionary, a list of toppings can be just one aspect of the pizza you're describing.

In the following example, two kinds of information are stored for each pizza: a type of crust and a list of toppings. The list of toppings is a value associated with the key `'toppings'`. To use the items in the list, we give the name of the dictionary and the key `'toppings'`, as we would any value in the dictionary. Instead of returning a single value, we get a list of toppings:

pizza.py

```
# Store information about a pizza being ordered.
pizza = {
    'crust': 'thick',
    'toppings': ['mushrooms', 'extra cheese'],
    }

# Summarize the order.
❶ print(f"You ordered a {pizza['crust']}-crust pizza "
    "with the following toppings:")

❷ for topping in pizza['toppings']:
    print(f"\t{topping}")
```

We begin with a dictionary that holds information about a pizza that has been ordered. One key in the dictionary is `'crust'`, and the associated value is the string `'thick'`. The next key, `'toppings'`, has a list as its value that stores all requested toppings. We summarize the order before building the pizza ❶. When you need to break up a long line in a `print()` call, choose an appropriate point at which to break the line being printed, and end the line with a quotation mark. Indent the next line, add an opening quotation mark, and continue the string. Python will automatically combine all of the strings it finds inside the parentheses. To print the toppings, we write a `for` loop ❷. To access the list of toppings, we use the key `'toppings'`, and Python grabs the list of toppings from the dictionary.

The following output summarizes the pizza that we plan to build:

```
You ordered a thick-crust pizza with the following toppings:
    mushrooms
    extra cheese
```

You can nest a list inside a dictionary anytime you want more than one value to be associated with a single key in a dictionary. In the earlier example of favorite programming languages, if we were to store each person's responses in a list, people could choose more than one favorite language.

When we loop through the dictionary, the value associated with each person would be a list of languages rather than a single language. Inside the dictionary's `for` loop, we use another `for` loop to run through the list of languages associated with each person:

favorite_languages.py

```
favorite_languages = {
    'jen': ['python', 'rust'],
    'sarah': ['c'],
    'edward': ['rust', 'go'],
    'phil': ['python', 'haskell'],
    }

❶ for name, languages in favorite_languages.items():
    print(f"\n{name.title()}'s favorite languages are:")
❷     for language in languages:
        print(f"\t{language.title()}")
```

The value associated with each name in `favorite_languages` is now a list. Note that some people have one favorite language and others have multiple favorites. When we loop through the dictionary ❶, we use the variable name `languages` to hold each value from the dictionary, because we know that each value will be a list. Inside the main dictionary loop, we use another `for` loop ❷ to run through each person's list of favorite languages. Now each person can list as many favorite languages as they like:

```
Jen's favorite languages are:
    Python
    Rust

Sarah's favorite languages are:
    C

Edward's favorite languages are:
    Rust
    Go

Phil's favorite languages are:
    Python
    Haskell
```

To refine this program even further, you could include an `if` statement at the beginning of the dictionary's `for` loop to see whether each person has more than one favorite language by examining the value of `len(languages)`. If a person has more than one favorite, the output would stay the same. If the person has only one favorite language, you could change the wording to reflect that. For example, you could say, "Sarah's favorite language is C."

NOTE *You should not nest lists and dictionaries too deeply. If you're nesting items much deeper than what you see in the preceding examples, or if you're working with someone else's code with significant levels of nesting, there's most likely a simpler way to solve the problem.*

A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the usernames as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username. In the following listing, we store three pieces of information about each user: their first name, last name, and location. We'll access this information by looping through the usernames and the dictionary of information associated with each username:

many_users.py

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },

    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },

    }

❶ for username, user_info in users.items():
❷     print(f"\nUsername: {username}")
❸     full_name = f"{user_info['first']} {user_info['last']}"
      location = user_info['location']

❹     print(f"\tFull name: {full_name.title()}")
      print(f"\tLocation: {location.title()}")
```

We first define a dictionary called `users` with two keys: one each for the usernames `'aeinstein'` and `'mcurie'`. The value associated with each key is a dictionary that includes each user's first name, last name, and location. Then, we loop through the `users` dictionary ❶. Python assigns each key to the variable `username`, and the dictionary associated with each username is assigned to the variable `user_info`. Once inside the main dictionary loop, we print the username ❷.

Then, we start accessing the inner dictionary ❸. The variable `user_info`, which contains the dictionary of user information, has three keys: `'first'`, `'last'`, and `'location'`. We use each key to generate a neatly formatted full name and location for each person, and then print a summary of what we know about each user ❹:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton
```

```
Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Notice that the structure of each user's dictionary is identical. Although not required by Python, this structure makes nested dictionaries easier to work with. If each user's dictionary had different keys, the code inside the `for` loop would be more complicated.

TRY IT YOURSELF

6-7. People: Start with the program you wrote for Exercise 6-1 (page 98). Make two new dictionaries representing different people, and store all three dictionaries in a list called `people`. Loop through your list of people. As you loop through the list, print everything you know about each person.

6-8. Pets: Make several dictionaries, where each dictionary represents a different pet. In each dictionary, include the kind of animal and the owner's name. Store these dictionaries in a list called `pets`. Next, loop through your list and as you do, print everything you know about each pet.

6-9. Favorite Places: Make a dictionary called `favorite_places`. Think of three names to use as keys in the dictionary, and store one to three favorite places for each person. To make this exercise a bit more interesting, ask some friends to name a few of their favorite places. Loop through the dictionary, and print each person's name and their favorite places.

6-10. Favorite Numbers: Modify your program from Exercise 6-2 (page 98) so each person can have more than one favorite number. Then print each person's name along with their favorite numbers.

6-11. Cities: Make a dictionary called `cities`. Use the names of three cities as keys in your dictionary. Create a dictionary of information about each city and include the country that the city is in, its approximate population, and one fact about that city. The keys for each city's dictionary should be something like `country`, `population`, and `fact`. Print the name of each city and all of the information you have stored about it.

6-12. Extensions: We're now working with examples that are complex enough that they can be extended in any number of ways. Use one of the example programs from this chapter, and extend it by adding new keys and values, changing the context of the program, or improving the formatting of the output.

Summary

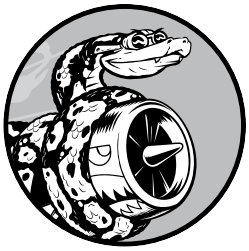
In this chapter, you learned how to define a dictionary and how to work with the information stored in a dictionary. You learned how to access and modify individual elements in a dictionary, and how to loop through all

of the information in a dictionary. You learned to loop through a dictionary's key-value pairs, its keys, and its values. You also learned how to nest multiple dictionaries in a list, nest lists in a dictionary, and nest a dictionary inside a dictionary.

In the next chapter you'll learn about `while` loops and how to accept input from people who are using your programs. This will be an exciting chapter, because you'll learn to make all of your programs interactive: they'll be able to respond to user input.

7

USER INPUT AND WHILE LOOPS



Most programs are written to solve an end user's problem. To do so, you usually need to get some information from the user. For example, say someone wants to find out whether they're old enough to vote. If you write a program to answer this question, you need to know the user's age before you can provide an answer. The program will need to ask the user to enter, or *input*, their age; once the program has this input, it can compare it to the voting age to determine if the user is old enough and then report the result.

In this chapter you'll learn how to accept user input so your program can then work with it. When your program needs a name, you'll be able to prompt the user for a name. When your program needs a list of names, you'll be able to prompt the user for a series of names. To do this, you'll use the `input()` function.

You'll also learn how to keep programs running as long as users want them to, so they can enter as much information as they need to; then, your

program can work with that information. You'll use Python's `while` loop to keep programs running as long as certain conditions remain true.

With the ability to work with user input and the ability to control how long your programs run, you'll be able to write fully interactive programs.

How the input() Function Works

The `input()` function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it assigns that input to a variable to make it convenient for you to work with.

For example, the following program asks the user to enter some text, then displays that message back to the user:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

The `input()` function takes one argument: the *prompt* that we want to display to the user, so they know what kind of information to enter. In this example, when Python runs the first line, the user sees the prompt `Tell me something, and I will repeat it back to you:` . The program waits while the user enters their response and continues after the user presses ENTER. The response is assigned to the variable `message`, then `print(message)` displays the input back to the user:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

NOTE *Some text editors won't run programs that prompt the user for input. You can use these editors to write programs that prompt for input, but you'll need to run these programs from a terminal. See "Running Python Programs from a Terminal" on page 11.*

Writing Clear Prompts

Each time you use the `input()` function, you should include a clear, easy-to-follow prompt that tells the user exactly what kind of information you're looking for. Any statement that tells the user what to enter should work. For example:

greeter.py

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

Add a space at the end of your prompts (after the colon in the preceding example) to separate the prompt from the user's response and to make it clear to your user where to enter their text. For example:

```
Please enter your name: Eric
Hello, Eric!
```

Sometimes you'll want to write a prompt that's longer than one line. For example, you might want to tell the user why you're asking for certain input. You can assign your prompt to a variable and pass that variable to the `input()` function. This allows you to build your prompt over several lines, then write a clean `input()` statement.

greeter.py

```
prompt = "If you share your name, we can personalize the messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print(f"\nHello, {name}!")
```

This example shows one way to build a multiline string. The first line assigns the first part of the message to the variable `prompt`. In the second line, the operator `+=` takes the string that was assigned to `prompt` and adds the new string onto the end.

The prompt now spans two lines, again with space after the question mark for clarity:

```
If you share your name, we can personalize the messages you see.
What is your first name? Eric

Hello, Eric!
```

Using int() to Accept Numerical Input

When you use the `input()` function, Python interprets everything the user enters as a string. Consider the following interpreter session, which asks for the user's age:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

The user enters the number `21`, but when we ask Python for the value of `age`, it returns `'21'`, the string representation of the numerical value entered. We know Python interpreted the input as a string because the number is now enclosed in quotes. If all you want to do is print the input, this works well. But if you try to use the input as a number, you'll get an error:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
❷ TypeError: '>=' not supported between instances of 'str' and 'int'
```

When you try to use the input to do a numerical comparison ❶, Python produces an error because it can't compare a string to an integer: the string `'21'` that's assigned to `age` can't be compared to the numerical value `18` ❷.

We can resolve this issue by using the `int()` function, which converts the input string to a numerical value. This allows the comparison to run successfully:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age = int(age)
>>> age >= 18
True
```

In this example, when we enter `21` at the prompt, Python interprets the number as a string, but the value is then converted to a numerical representation by `int()` ❶. Now Python can run the conditional test: it compares `age` (which now represents the numerical value 21) and `18` to see if `age` is greater than or equal to 18. This test evaluates to `True`.

How do you use the `int()` function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

rollercoaster.py

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

The program can compare `height` to `48` because `height = int(height)` converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 48, we tell the user that they're tall enough:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

When you use numerical input to do calculations and comparisons, be sure to convert the input value to a numerical representation first.

The Modulo Operator

A useful tool for working with numerical information is the *modulo operator* (`%`), which divides one number by another number and returns the remainder:

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

The modulo operator doesn't tell you how many times one number fits into another; it only tells you what the remainder is.

When one number is divisible by another number, the remainder is 0, so the modulo operator always returns 0. You can use this fact to determine if a number is even or odd:

even_or_odd.py

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

Even numbers are always divisible by two, so if the modulo of a number and two is zero (here, `if number % 2 == 0`) the number is even. Otherwise, it's odd.

```
Enter a number, and I'll tell you if it's even or odd: 42

The number 42 is even.
```

TRY IT YOURSELF

7-1. Rental Car: Write a program that asks the user what kind of rental car they would like. Print a message about that car, such as "Let me see if I can find you a Subaru."

7-2. Restaurant Seating: Write a program that asks the user how many people are in their dinner group. If the answer is more than eight, print a message saying they'll have to wait for a table. Otherwise, report that their table is ready.

7-3. Multiples of Ten: Ask the user for a number, and then report whether the number is a multiple of 10 or not.

Introducing while Loops

The `for` loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the `while` loop runs as long as, or *while*, a certain condition is true.

The while Loop in Action

You can use a `while` loop to count up through a series of numbers. For example, the following `while` loop counts from 1 to 5:

counting.py

```
current_number = 1
while current_number <= 5:
```

```
    print(current_number)
    current_number += 1
```

In the first line, we start counting from 1 by assigning `current_number` the value 1. The `while` loop is then set to keep running as long as the value of `current_number` is less than or equal to 5. The code inside the loop prints the value of `current_number` and then adds 1 to that value with `current_number += 1`. (The `+=` operator is shorthand for `current_number = current_number + 1`.)

Python repeats the loop as long as the condition `current_number <= 5` is true. Because 1 is less than 5, Python prints `1` and then adds 1, making the current number `2`. Because 2 is less than 5, Python prints `2` and adds 1 again, making the current number `3`, and so on. Once the value of `current_number` is greater than 5, the loop stops running and the program ends:

```
1
2
3
4
5
```

The programs you use every day most likely contain `while` loops. For example, a game needs a `while` loop to keep running as long as you want to keep playing, and so it can stop running as soon as you ask it to quit. Programs wouldn't be fun to use if they stopped running before we told them to or kept running even after we wanted to quit, so `while` loops are quite useful.

Letting the User Choose When to Quit

We can make the *parrot.py* program run as long as the user wants by putting most of the program inside a `while` loop. We'll define a *quit value* and then keep the program running as long as the user has not entered the quit value:

parrot.py

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)
    print(message)
```

We first define a prompt that tells the user their two options: entering a message or entering the quit value (in this case, `'quit'`). Then we set up a variable `message` to keep track of whatever value the user enters. We define `message` as an empty string, `""`, so Python has something to check the first time it reaches the `while` line. The first time the program runs and Python reaches the `while` statement, it needs to compare the value of `message` to `'quit'`, but no user input has been entered yet. If Python has nothing to compare, it won't be able to continue running the program. To solve this

problem, we make sure to give `message` an initial value. Although it's just an empty string, it will make sense to Python and allow it to perform the comparison that makes the `while` loop work. This `while` loop runs as long as the value of `message` is not `'quit'`.

The first time through the loop, `message` is just an empty string, so Python enters the loop. At `message = input(prompt)`, Python displays the prompt and waits for the user to enter their input. Whatever they enter is assigned to `message` and printed; then, Python reevaluates the condition in the `while` statement. As long as the user has not entered the word `'quit'`, the prompt is displayed again and Python waits for more input. When the user finally enters `'quit'`, Python stops executing the `while` loop and the program ends:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

This program works well, except that it prints the word `'quit'` as if it were an actual message. A simple `if` test fixes this:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Now the program makes a quick check before displaying the message and only prints the message if it does not match the quit value:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Using a Flag

In the previous example, we had the program perform certain tasks while a given condition was true. But what about more complicated programs in which many different events could cause the program to stop running?

For example, in a game, several different events can end the game. When the player runs out of ships, their time runs out, or the cities they were supposed to protect are all destroyed, the game should end. It needs to end if any one of these events happens. If many possible events might occur to stop the program, trying to test all these conditions in one `while` statement becomes complicated and difficult.

For a program that should run only as long as many conditions are true, you can define one variable that determines whether or not the entire program is active. This variable, called a *flag*, acts as a signal to the program. We can write our programs so they run while the flag is set to `True` and stop running when any of several events sets the value of the flag to `False`. As a result, our overall `while` statement needs to check only one condition: whether the flag is currently `True`. Then, all our other tests (to see if an event has occurred that should set the flag to `False`) can be neatly organized in the rest of the program.

Let's add a flag to *parrot.py* from the previous section. This flag, which we'll call `active` (though you can call it anything), will monitor whether or not the program should continue running:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

active = True
❶ while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

We set the variable `active` to `True` so the program starts in an active state. Doing so makes the `while` statement simpler because no comparison is made in the `while` statement itself; the logic is taken care of in other parts of the program. As long as the `active` variable remains `True`, the loop will continue running ❶.

In the `if` statement inside the `while` loop, we check the value of `message` once the user enters their input. If the user enters `'quit'`, we set `active` to `False`, and the `while` loop stops. If the user enters anything other than `'quit'`, we print their input as a message.

This program has the same output as the previous example where we placed the conditional test directly in the `while` statement. But now that we

have a flag to indicate whether the overall program is in an active state, it would be easy to add more tests (such as `elif` statements) for events that should cause `active` to become `False`. This is useful in complicated programs like games, in which there may be many events that should each make the program stop running. When any of these events causes the active flag to become `False`, the main game loop will exit, a *Game Over* message can be displayed, and the player can be given the option to play again.

Using break to Exit a Loop

To exit a `while` loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the `break` statement. The `break` statement directs the flow of your program; you can use it to control which lines of code are executed and which aren't, so the program only executes code that you want it to, when you want it to.

For example, consider a program that asks the user about places they've visited. We can stop the `while` loop in this program by calling `break` as soon as the user enters the `'quit'` value:

cities.py

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "

❶ while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

A loop that starts with `while True` ❶ will run forever unless it reaches a `break` statement. The loop in this program continues asking the user to enter the names of cities they've been to until they enter `'quit'`. When they enter `'quit'`, the `break` statement runs, causing Python to exit the loop:

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

NOTE *You can use the `break` statement in any of Python's loops. For example, you could use `break` to quit a `for` loop that's working through a list or a dictionary.*

Using continue in a Loop

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the `continue` statement to return to the beginning of the loop, based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

counting.py

```
current_number = 0
while current_number < 10:
❶    current_number += 1
    if current_number % 2 == 0:
        continue

    print(current_number)
```

First, we set `current_number` to 0. Because it's less than 10, Python enters the `while` loop. Once inside the loop, we increment the count by 1 ❶, so `current_number` is 1. The `if` statement then checks the modulo of `current_number` and 2. If the modulo is 0 (which means `current_number` is divisible by 2), the `continue` statement tells Python to ignore the rest of the loop and return to the beginning. If the current number is not divisible by 2, the rest of the loop is executed and Python prints the current number:

```
1
3
5
7
9
```

Avoiding Infinite Loops

Every `while` loop needs a way to stop running so it won't continue to run forever. For example, this counting loop should count from 1 to 5:

counting.py

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

However, if you accidentally omit the line `x += 1`, the loop will run forever:

```
# This loop runs forever!
x = 1
while x <= 5:
    print(x)
```

Now the value of `x` will start at `1` but never change. As a result, the conditional test `x <= 5` will always evaluate to `True` and the `while` loop will run forever, printing a series of 1s, like this:

```
1
1
1
1
--snip--
```

Every programmer accidentally writes an infinite `while` loop from time to time, especially when a program's loops have subtle exit conditions. If your program gets stuck in an infinite loop, press CTRL-C or just close the terminal window displaying your program's output.

To avoid writing infinite loops, test every `while` loop and make sure the loop stops when you expect it to. If you want your program to end when the user enters a certain input value, run the program and enter that value. If the program doesn't end, scrutinize the way your program handles the value that should cause the loop to exit. Make sure at least one part of the program can make the loop's condition `False` or cause it to reach a `break` statement.

NOTE *VS Code, like many editors, displays output in an embedded terminal window. To cancel an infinite loop, make sure you click in the output area of the editor before pressing CTRL-C.*

TRY IT YOURSELF

7-4. Pizza Toppings: Write a loop that prompts the user to enter a series of pizza toppings until they enter a `'quit'` value. As they enter each topping, print a message saying you'll add that topping to their pizza.

7-5. Movie Tickets: A movie theater charges different ticket prices depending on a person's age. If a person is under the age of 3, the ticket is free; if they are between 3 and 12, the ticket is $10; and if they are over age 12, the ticket is $15. Write a loop in which you ask users their age, and then tell them the cost of their movie ticket.

7-6. Three Exits: Write different versions of either Exercise 7-4 or 7-5 that do each of the following at least once:

- Use a conditional test in the `while` statement to stop the loop.
- Use an `active` variable to control how long the loop runs.
- Use a `break` statement to exit the loop when the user enters a `'quit'` value.

7-7. Infinity: Write a loop that never ends, and run it. (To end the loop, press CTRL-C or close the window displaying the output.)

Using a while Loop with Lists and Dictionaries

So far, we've worked with only one piece of user information at a time. We received the user's input and then printed the input or a response to it. The next time through the `while` loop, we'd receive another input value and respond to that. But to keep track of many users and pieces of information, we'll need to use lists and dictionaries with our `while` loops.

A `for` loop is effective for looping through a list, but you shouldn't modify a list inside a `for` loop because Python will have trouble keeping track of the items in the list. To modify a list as you work through it, use a `while` loop. Using `while` loops with lists and dictionaries allows you to collect, store, and organize lots of input to examine and report on later.

Moving Items from One List to Another

Consider a list of newly registered but unverified users of a website. After we verify these users, how can we move them to a separate list of confirmed users? One way would be to use a `while` loop to pull users from the list of unconfirmed users as we verify them and then add them to a separate list of confirmed users. Here's what that code might look like:

confirmed_users.py

```
# Start with users that need to be verified,
#  and an empty list to hold confirmed users.
❶ unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verify each user until there are no more unconfirmed users.
#  Move each verified user into the list of confirmed users.
❷ while unconfirmed_users:
❸     current_user = unconfirmed_users.pop()

      print(f"Verifying user: {current_user.title()}")
❹     confirmed_users.append(current_user)

# Display all confirmed users.
print("\nThe following users have been confirmed:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

We begin with a list of unconfirmed users ❶ (Alice, Brian, and Candace) and an empty list to hold confirmed users. The `while` loop runs as long as the list `unconfirmed_users` is not empty ❷. Within this loop, the `pop()` method removes unverified users one at a time from the end of `unconfirmed_users` ❸. Because Candace is last in the `unconfirmed_users` list, her name will be the first to be removed, assigned to `current_user`, and added to the `confirmed_users` list ❹. Next is Brian, then Alice.

We simulate confirming each user by printing a verification message and then adding them to the list of confirmed users. As the list of unconfirmed users shrinks, the list of confirmed users grows. When the list of

unconfirmed users is empty, the loop stops and the list of confirmed users is printed:

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Removing All Instances of Specific Values from a List

In Chapter 3, we used `remove()` to remove a specific value from a list. The `remove()` function worked because the value we were interested in appeared only once in the list. But what if you want to remove all instances of a value from a list?

Say you have a list of pets with the value `'cat'` repeated several times. To remove all instances of that value, you can run a `while` loop until `'cat'` is no longer in the list, as shown here:

pets.py

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

We start with a list containing multiple instances of `'cat'`. After printing the list, Python enters the `while` loop because it finds the value `'cat'` in the list at least once. Once inside the loop, Python removes the first instance of `'cat'`, returns to the `while` line, and then reenters the loop when it finds that `'cat'` is still in the list. It removes each instance of `'cat'` until the value is no longer in the list, at which point Python exits the loop and prints the list again:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Filling a Dictionary with User Input

You can prompt for as much input as you need in each pass through a `while` loop. Let's make a polling program in which each pass through the loop prompts for the participant's name and response. We'll store the data we gather in a dictionary, because we want to connect each response with a particular user:

mountain_poll.py

```
responses = {}
# Set a flag to indicate that polling is active.
polling_active = True
```

```
while polling_active:
    # Prompt for the person's name and response.
❶   name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday? ")

    # Store the response in the dictionary.
❷   responses[name] = response

    # Find out if anyone else is going to take the poll.
❸   repeat = input("Would you like to let another person respond? (yes/ no) ")
    if repeat == 'no':
        polling_active = False

# Polling is complete. Show the results.
print("\n--- Poll Results ---")
❹ for name, response in responses.items():
    print(f"{name} would like to climb {response}.")
```

The program first defines an empty dictionary (`responses`) and sets a flag (`polling_active`) to indicate that polling is active. As long as `polling_active` is `True`, Python will run the code in the `while` loop.

Within the loop, the user is prompted to enter their name and a mountain they'd like to climb ❶. That information is stored in the `responses` dictionary ❷, and the user is asked whether or not to keep the poll running ❸. If they enter `yes`, the program enters the `while` loop again. If they enter `no`, the `polling_active` flag is set to `False`, the `while` loop stops running, and the final code block ❹ displays the results of the poll.

If you run this program and enter sample responses, you should see output like this:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Eric would like to climb Denali.
Lynn would like to climb Devil's Thumb.
```

TRY IT YOURSELF

7-8. Deli: Make a list called `sandwich_orders` and fill it with the names of various sandwiches. Then make an empty list called `finished_sandwiches`. Loop through the list of sandwich orders and print a message for each order, such as `I made your tuna sandwich`. As each sandwich is made, move it to the list of finished sandwiches. After all the sandwiches have been made, print a message listing each sandwich that was made.

7-9. No Pastrami: Using the list `sandwich_orders` from Exercise 7-8, make sure the sandwich `'pastrami'` appears in the list at least three times. Add code near the beginning of your program to print a message saying the deli has run out of pastrami, and then use a `while` loop to remove all occurrences of `'pastrami'` from `sandwich_orders`. Make sure no pastrami sandwiches end up in `finished_sandwiches`.

7-10. Dream Vacation: Write a program that polls users about their dream vacation. Write a prompt similar to *If you could visit one place in the world, where would you go?* Include a block of code that prints the results of the poll.

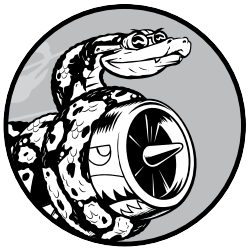
Summary

In this chapter, you learned how to use `input()` to allow users to provide their own information in your programs. You learned to work with both text and numerical input and how to use `while` loops to make your programs run as long as your users want them to. You saw several ways to control the flow of a `while` loop by setting an `active` flag, using the `break` statement, and using the `continue` statement. You learned how to use a `while` loop to move items from one list to another and how to remove all instances of a value from a list. You also learned how `while` loops can be used with dictionaries.

In Chapter 8 you'll learn about functions. *Functions* allow you to break your programs into small parts, each of which does one specific job. You can call a function as many times as you want, and you can store your functions in separate files. By using functions, you'll be able to write more efficient code that's easier to troubleshoot and maintain and that can be reused in many different programs.

8

FUNCTIONS



In this chapter you'll learn to write *functions*, which are named blocks of code designed to do one specific job. When you want to perform a particular task that you've defined in a function, you *call* the function responsible for it. If you need to perform that task multiple times throughout your program, you don't need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function. You'll find that using functions makes your programs easier to write, read, test, and fix.

In this chapter you'll also learn a variety of ways to pass information to functions. You'll learn how to write certain functions whose primary job is to display information and other functions designed to process data and return a value or set of values. Finally, you'll learn to store functions in separate files called *modules* to help organize your main program files.

Defining a Function

Here's a simple function named `greet_user()` that prints a greeting:

greeter.py

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

This example shows the simplest structure of a function. The first line uses the keyword `def` to inform Python that you're defining a function. This is the *function definition*, which tells Python the name of the function and, if applicable, what kind of information the function needs to do its job. The parentheses hold that information. In this case, the name of the function is `greet_user()`, and it needs no information to do its job, so its parentheses are empty. (Even so, the parentheses are required.) Finally, the definition ends in a colon.

Any indented lines that follow `def greet_user():` make up the *body* of the function. The text on the second line is a comment called a *docstring*, which describes what the function does. When Python generates documentation for the functions in your programs, it looks for a string immediately after the function's definition. These strings are usually enclosed in triple quotes, which lets you write multiple lines.

The line `print("Hello!")` is the only line of actual code in the body of this function, so `greet_user()` has just one job: `print("Hello!")`.

When you want to use this function, you have to call it. A *function call* tells Python to execute the code in the function. To *call* a function, you write the name of the function, followed by any necessary information in parentheses. Because no information is needed here, calling our function is as simple as entering `greet_user()`. As expected, it prints `Hello!`:

```
Hello!
```

Passing Information to a Function

If you modify the function `greet_user()` slightly, it can greet the user by name. For the function to do this, you enter `username` in the parentheses of the function's definition at `def greet_user()`. By adding `username` here, you allow the function to accept any value of `username` you specify. The function now expects you to provide a value for `username` each time you call it. When you call `greet_user()`, you can pass it a name, such as `'jesse'`, inside the parentheses:

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

Entering `greet_user('jesse')` calls `greet_user()` and gives the function the information it needs to execute the `print()` call. The function accepts the name you passed it and displays the greeting for that name:

```
Hello, Jesse!
```

Likewise, entering `greet_user('sarah')` calls `greet_user()`, passes it `'sarah'`, and prints `Hello, Sarah!` You can call `greet_user()` as often as you want and pass it any name you want to produce a predictable output every time.

Arguments and Parameters

In the preceding `greet_user()` function, we defined `greet_user()` to require a value for the variable `username`. Once we called the function and gave it the information (a person's name), it printed the right greeting.

The variable `username` in the definition of `greet_user()` is an example of a *parameter*, a piece of information the function needs to do its job. The value `'jesse'` in `greet_user('jesse')` is an example of an argument. An *argument* is a piece of information that's passed from a function call to a function. When we call the function, we place the value we want the function to work with in parentheses. In this case the argument `'jesse'` was passed to the function `greet_user()`, and the value was assigned to the parameter `username`.

NOTE *People sometimes speak of arguments and parameters interchangeably. Don't be surprised if you see the variables in a function definition referred to as arguments or the variables in a function call referred to as parameters.*

TRY IT YOURSELF

8-1. Message: Write a function called `display_message()` that prints one sentence telling everyone what you are learning about in this chapter. Call the function, and make sure the message displays correctly.

8-2. Favorite Book: Write a function called `favorite_book()` that accepts one parameter, `title`. The function should print a message, such as `One of my favorite books is Alice in Wonderland`. Call the function, making sure to include a book title as an argument in the function call.

Passing Arguments

Because a function definition can have multiple parameters, a function call may need multiple arguments. You can pass arguments to your functions in a number of ways. You can use *positional arguments*, which need to be in the same order the parameters were written; *keyword arguments*, where each argument consists of a variable name and a value; and lists and dictionaries of values. Let's look at each of these in turn.

Positional Arguments

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called *positional arguments*.

To see how this works, consider a function that displays information about pets. The function tells us what kind of animal each pet is and the pet's name, as shown here:

pets.py

```
❶ def describe_pet(animal_type, pet_name):
      """Display information about a pet."""
      print(f"\nI have a {animal_type}.")
      print(f"My {animal_type}'s name is {pet_name.title()}.")

❷ describe_pet('hamster', 'harry')
```

The definition shows that this function needs a type of animal and the animal's name ❶. When we call `describe_pet()`, we need to provide an animal type and a name, in that order. For example, in the function call, the argument `'hamster'` is assigned to the parameter `animal_type` and the argument `'harry'` is assigned to the parameter `pet_name` ❷. In the function body, these two parameters are used to display information about the pet being described.

The output describes a hamster named Harry:

```
I have a hamster.
My hamster's name is Harry.
```

Multiple Function Calls

You can call a function as many times as needed. Describing a second, different pet requires just one more call to `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

In this second function call, we pass `describe_pet()` the arguments `'dog'` and `'willie'`. As with the previous set of arguments we used, Python matches `'dog'` with the parameter `animal_type` and `'willie'` with the parameter `pet_name`. As before, the function does its job, but this time it prints values for a dog named Willie. Now we have a hamster named Harry and a dog named Willie:

```
I have a hamster.
My hamster's name is Harry.
```

```
I have a dog.
My dog's name is Willie.
```

Calling a function multiple times is a very efficient way to work. The code describing a pet is written once in the function. Then, anytime you want to describe a new pet, you call the function with the new pet's information. Even if the code for describing a pet were to expand to 10 lines, you could still describe a new pet in just one line by calling the function again.

Order Matters in Positional Arguments

You can get unexpected results if you mix up the order of the arguments in a function call when using positional arguments:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

In this function call, we list the name first and the type of animal second. Because the argument `'harry'` is listed first this time, that value is assigned to the parameter `animal_type`. Likewise, `'hamster'` is assigned to `pet_name`. Now we have a "harry" named "Hamster":

```
I have a harry.
My harry's name is Hamster.
```

If you get funny results like this, check to make sure the order of the arguments in your function call matches the order of the parameters in the function's definition.

Keyword Arguments

A *keyword argument* is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion (you won't end up with a harry named Hamster). Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

Let's rewrite *pets.py* using keyword arguments to call `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

The function `describe_pet()` hasn't changed. But when we call the function, we explicitly tell Python which parameter each argument should be matched with. When Python reads the function call, it knows to assign the argument `'hamster'` to the parameter `animal_type` and the argument `'harry'` to `pet_name`. The output correctly shows that we have a hamster named Harry.

The order of keyword arguments doesn't matter because Python knows where each value should go. The following two function calls are equivalent:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

NOTE *When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.*

Default Values

When writing a function, you can define a *default value* for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value. So when you define a default value for a parameter, you can exclude the corresponding argument you'd usually write in the function call. Using default values can simplify your function calls and clarify the ways your functions are typically used.

For example, if you notice that most of the calls to `describe_pet()` are being used to describe dogs, you can set the default value of `animal_type` to `'dog'`. Now anyone calling `describe_pet()` for a dog can omit that information:

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

We changed the definition of `describe_pet()` to include a default value, `'dog'`, for `animal_type`. Now when the function is called with no `animal_type` specified, Python knows to use the value `'dog'` for this parameter:

```
I have a dog.
My dog's name is Willie.
```

Note that the order of the parameters in the function definition had to be changed. Because the default value makes it unnecessary to specify a type of animal as an argument, the only argument left in the function call is the pet's name. Python still interprets this as a positional argument, so if the function is called with just a pet's name, that argument will match up with the first parameter listed in the function's definition. This is the reason the first parameter needs to be `pet_name`.

The simplest way to use this function now is to provide just a dog's name in the function call:

```
describe_pet('willie')
```

This function call would have the same output as the previous example. The only argument provided is `'willie'`, so it is matched up with the first parameter in the definition, `pet_name`. Because no argument is provided for `animal_type`, Python uses the default value `'dog'`.

To describe an animal other than a dog, you could use a function call like this:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Because an explicit argument for `animal_type` is provided, Python will ignore the parameter's default value.

NOTE *When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly.*

Equivalent Function Calls

Because positional arguments, keyword arguments, and default values can all be used together, you'll often have several equivalent ways to call a function. Consider the following definition for `describe_pet()` with one default value provided:

```
def describe_pet(pet_name, animal_type='dog'):
```

With this definition, an argument always needs to be provided for `pet_name`, and this value can be provided using the positional or keyword format. If the animal being described is not a dog, an argument for `animal_type` must be included in the call, and this argument can also be specified using the positional or keyword format.

All of the following calls would work for this function:

```
# A dog named Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# A hamster named Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Each of these function calls would have the same output as the previous examples.

It doesn't really matter which calling style you use. As long as your function calls produce the output you want, just use the style you find easiest to understand.

Avoiding Argument Errors

When you start to use functions, don't be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work. For example, here's what happens if we try to call `describe_pet()` with no arguments:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet()
```

Python recognizes that some information is missing from the function call, and the traceback tells us that:

```
Traceback (most recent call last):
❶  File "pets.py", line 6, in <module>
❷    describe_pet()
     ^^^^^^^^^^^^^^
❸ TypeError: describe_pet() missing 2 required positional arguments:
     'animal_type' and 'pet_name'
```

The traceback first tells us the location of the problem ❶, allowing us to look back and see that something went wrong in our function call. Next, the offending function call is written out for us to see ❷. Last, the traceback tells us the call is missing two arguments and reports the names of the missing arguments ❸. If this function were in a separate file, we could probably rewrite the call correctly without having to open that file and read the function code.

Python is helpful in that it reads the function's code for us and tells us the names of the arguments we need to provide. This is another motivation for giving your variables and functions descriptive names. If you do, Python's error messages will be more useful to you and anyone else who might use your code.

If you provide too many arguments, you should get a similar traceback that can help you correctly match your function call to the function definition.

TRY IT YOURSELF

8-3. T-Shirt: Write a function called `make_shirt()` that accepts a size and the text of a message that should be printed on the shirt. The function should print a sentence summarizing the size of the shirt and the message printed on it.

Call the function once using positional arguments to make a shirt. Call the function a second time using keyword arguments.

8-4. Large Shirts: Modify the `make_shirt()` function so that shirts are large by default with a message that reads *I love Python*. Make a large shirt and a medium shirt with the default message, and a shirt of any size with a different message.

8-5. Cities: Write a function called `describe_city()` that accepts the name of a city and its country. The function should print a simple sentence, such as `Reykjavik is in Iceland`. Give the parameter for the country a default value. Call your function for three different cities, at least one of which is not in the default country.

Return Values

A function doesn't always have to display its output directly. Instead, it can process some data and then return a value or set of values. The value the function returns is called a *return value*. The `return` statement takes a value from inside a function and sends it back to the line that called the function. Return values allow you to move much of your program's grunt work into functions, which can simplify the body of your program.

Returning a Simple Value

Let's look at a function that takes a first and last name, and returns a neatly formatted full name:

formatted_name.py

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
❶   full_name = f"{first_name} {last_name}"
❷   return full_name.title()

❸ musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

The definition of `get_formatted_name()` takes as parameters a first and last name. The function combines these two names, adds a space between them, and assigns the result to `full_name` ❶. The value of `full_name` is converted to title case, and then returned to the calling line ❷.

When you call a function that returns a value, you need to provide a variable that the return value can be assigned to. In this case, the returned value is assigned to the variable `musician` ❸. The output shows a neatly formatted name made up of the parts of a person's name:

```
Jimi Hendrix
```

This might seem like a lot of work to get a neatly formatted name when we could have just written:

```
print("Jimi Hendrix")
```

However, when you consider working with a large program that needs to store many first and last names separately, functions like `get_formatted_name()` become very useful. You store first and last names separately and then call this function whenever you want to display a full name.

Making an Argument Optional

Sometimes it makes sense to make an argument optional, so that people using the function can choose to provide extra information only if they want to. You can use default values to make an argument optional.

For example, say we want to expand `get_formatted_name()` to handle middle names as well. A first attempt to include middle names might look like this:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

This function works when given a first, middle, and last name. The function takes in all three parts of a name and then builds a string out of them. The function adds spaces where appropriate and converts the full name to title case:

```
John Lee Hooker
```

But middle names aren't always needed, and this function as written would not work if you tried to call it with only a first name and a last name. To make the middle name optional, we can give the `middle_name` argument an empty default value and ignore the argument unless the user provides a value. To make `get_formatted_name()` work without a middle name, we set the default value of `middle_name` to an empty string and move it to the end of the list of parameters:

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Return a full name, neatly formatted."""
❶   if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
❷   else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

❸ musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

In this example, the name is built from three possible parts. Because there's always a first and last name, these parameters are listed first in the function's definition. The middle name is optional, so it's listed last in the definition, and its default value is an empty string.

In the body of the function, we check to see if a middle name has been provided. Python interprets non-empty strings as `True`, so the conditional test `if middle_name` evaluates to `True` if a middle name argument is in the function call ❶. If a middle name is provided, the first, middle, and last names are combined to form a full name. This name is then changed to title case and returned to the function call line, where it's assigned to the variable `musician` and printed. If no middle name is provided, the empty string fails the `if` test and the `else` block runs ❷. The full name is made with just a first and last name, and the formatted name is returned to the calling line where it's assigned to `musician` and printed.

Calling this function with a first and last name is straightforward. If we're using a middle name, however, we have to make sure the middle name is the last argument passed so Python will match up the positional arguments correctly ❸.

This modified version of our function works for people with just a first and last name, and it works for people who have a middle name as well:

```
Jimi Hendrix
John Lee Hooker
```

Optional values allow functions to handle a wide range of use cases while letting function calls remain as simple as possible.

Returning a Dictionary

A function can return any kind of value you need it to, including more complicated data structures like lists and dictionaries. For example, the following function takes in parts of a name and returns a dictionary representing a person:

person.py

```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person."""
❶   person = {'first': first_name, 'last': last_name}
❷   return person

musician = build_person('jimi', 'hendrix')
❸ print(musician)
```

The function `build_person()` takes in a first and last name, and puts these values into a dictionary ❶. The value of `first_name` is stored with the key `'first'`, and the value of `last_name` is stored with the key `'last'`. Then, the entire dictionary representing the person is returned ❷. The return value is printed ❸ with the original two pieces of textual information now stored in a dictionary:

```
{'first': 'jimi', 'last': 'hendrix'}
```

This function takes in simple textual information and puts it into a more meaningful data structure that lets you work with the information beyond just printing it. The strings `'jimi'` and `'hendrix'` are now labeled as a first name and last name. You can easily extend this function to accept optional values like a middle name, an age, an occupation, or any other information you want to store about a person. For example, the following change allows you to store a person's age as well:

```
def build_person(first_name, last_name, age=None):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

We add a new optional parameter `age` to the function definition and assign the parameter the special value `None`, which is used when a variable has no specific value assigned to it. You can think of `None` as a placeholder value. In conditional tests, `None` evaluates to `False`. If the function call includes a value for `age`, that value is stored in the dictionary. This function always stores a person's name, but it can also be modified to store any other information you want about a person.

Using a Function with a while Loop

You can use functions with all the Python structures you've learned about so far. For example, let's use the `get_formatted_name()` function with a `while` loop to greet users more formally. Here's a first attempt at greeting people using their first and last names:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# This is an infinite loop!
while True:
❶   print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

For this example, we use a simple version of `get_formatted_name()` that doesn't involve middle names. The `while` loop asks the user to enter their name, and we prompt for their first and last name separately ❶.

But there's one problem with this `while` loop: We haven't defined a quit condition. Where do you put a quit condition when you ask for a series of

inputs? We want the user to be able to quit as easily as possible, so each prompt should offer a way to quit. The `break` statement offers a straightforward way to exit the loop at either prompt:

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

We add a message that informs the user how to quit, and then we break out of the loop if the user enters the quit value at either prompt. Now the program will continue greeting people until someone enters `q` for either name:

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes

Hello, Eric Matthes!

Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

TRY IT YOURSELF

8-6. City Names: Write a function called `city_country()` that takes in the name of a city and its country. The function should return a string formatted like this:

```
"Santiago, Chile"
```

Call your function with at least three city-country pairs, and print the values that are returned.

(continued)

8-7. Album: Write a function called `make_album()` that builds a dictionary describing a music album. The function should take in an artist name and an album title, and it should return a dictionary containing these two pieces of information. Use the function to make three dictionaries representing different albums. Print each return value to show that the dictionaries are storing the album information correctly.

Use `None` to add an optional parameter to `make_album()` that allows you to store the number of songs on an album. If the calling line includes a value for the number of songs, add that value to the album's dictionary. Make at least one new function call that includes the number of songs on an album.

8-8. User Albums: Start with your program from Exercise 8-7. Write a `while` loop that allows users to enter an album's artist and title. Once you have that information, call `make_album()` with the user's input and print the dictionary that's created. Be sure to include a quit value in the `while` loop.

Passing a List

You'll often find it useful to pass a list to a function, whether it's a list of names, numbers, or more complex objects, such as dictionaries. When you pass a list to a function, the function gets direct access to the contents of the list. Let's use functions to make working with lists more efficient.

Say we have a list of users and want to print a greeting to each. The following example sends a list of names to a function called `greet_users()`, which greets each person in the list individually:

greet_users.py

```
def greet_users(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

We define `greet_users()` so it expects a list of names, which it assigns to the parameter `names`. The function loops through the list it receives and prints a greeting to each user. Outside of the function, we define a list of users and then pass the list `usernames` to `greet_users()` in the function call:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

This is the output we wanted. Every user sees a personalized greeting, and you can call the function anytime you want to greet a specific set of users.

Modifying a List in a Function

When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function's body are permanent, allowing you to work efficiently even when you're dealing with large amounts of data.

Consider a company that creates 3D printed models of designs that users submit. Designs that need to be printed are stored in a list, and after being printed they're moved to a separate list. The following code does this without using functions:

printing_models.py

```
# Start with some designs that need to be printed.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simulate printing each design, until none are left.
#  Move each design to completed_models after printing.
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

# Display all completed models.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

This program starts with a list of designs that need to be printed and an empty list called `completed_models` that each design will be moved to after it has been printed. As long as designs remain in `unprinted_designs`, the `while` loop simulates printing each design by removing a design from the end of the list, storing it in `current_design`, and displaying a message that the current design is being printed. It then adds the design to the list of completed models. When the loop is finished running, a list of the designs that have been printed is displayed:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

We can reorganize this code by writing two functions, each of which does one specific job. Most of the code won't change; we're just structuring it more carefully. The first function will handle printing the designs, and the second will summarize the prints that have been made:

```
❶ def print_models(unprinted_designs, completed_models):
    """
    Simulate printing each design, until none are left.
```

```
    Move each design to completed_models after printing.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

❷ def show_completed_models(completed_models):
    """Show all the models that were printed."""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

We define the function `print_models()` with two parameters: a list of designs that need to be printed and a list of completed models ❶. Given these two lists, the function simulates printing each design by emptying the list of unprinted designs and filling up the list of completed models. We then define the function `show_completed_models()` with one parameter: the list of completed models ❷. Given this list, `show_completed_models()` displays the name of each model that was printed.

This program has the same output as the version without functions, but the code is much more organized. The code that does most of the work has been moved to two separate functions, which makes the main part of the program easier to understand. Look at the body of the program and notice how easily you can follow what's happening:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

We set up a list of unprinted designs and an empty list that will hold the completed models. Then, because we've already defined our two functions, all we have to do is call them and pass them the right arguments. We call `print_models()` and pass it the two lists it needs; as expected, `print_models()` simulates printing the designs. Then we call `show_completed_models()` and pass it the list of completed models so it can report the models that have been printed. The descriptive function names allow others to read this code and understand it, even without comments.

This program is easier to extend and maintain than the version without functions. If we need to print more designs later on, we can simply call

`print_models()` again. If we realize the printing code needs to be modified, we can change the code once, and our changes will take place everywhere the function is called. This technique is more efficient than having to update code separately in several places in the program.

This example also demonstrates the idea that every function should have one specific job. The first function prints each design, and the second displays the completed models. This is more beneficial than using one function to do both jobs. If you're writing a function and notice the function is doing too many different tasks, try to split the code into two functions. Remember that you can always call a function from another function, which can be helpful when splitting a complex task into a series of steps.

Preventing a Function from Modifying a List

Sometimes you'll want to prevent a function from modifying a list. For example, say that you start with a list of unprinted designs and write a function to move them to a list of completed models, as in the previous example. You may decide that even though you've printed all the designs, you want to keep the original list of unprinted designs for your records. But because you moved all the design names out of `unprinted_designs`, the list is now empty, and the empty list is the only version you have; the original is gone. In this case, you can address this issue by passing the function a copy of the list, not the original. Any changes the function makes to the list will affect only the copy, leaving the original list intact.

You can send a copy of a list to a function like this:

```
function_name(list_name[:])
```

The slice notation `[:]` makes a copy of the list to send to the function. If we didn't want to empty the list of unprinted designs in *printing_models.py*, we could call `print_models()` like this:

```
print_models(unprinted_designs[:], completed_models)
```

The function `print_models()` can do its work because it still receives the names of all unprinted designs. But this time it uses a copy of the original unprinted designs list, not the actual `unprinted_designs` list. The list `completed_models` will fill up with the names of printed models like it did before, but the original list of unprinted designs will be unaffected by the function.

Even though you can preserve the contents of a list by passing a copy of it to your functions, you should pass the original list to functions unless you have a specific reason to pass a copy. It's more efficient for a function to work with an existing list, because this avoids using the time and memory needed to make a separate copy. This is especially true when working with large lists.

TRY IT YOURSELF

8-9. Messages: Make a list containing a series of short text messages. Pass the list to a function called `show_messages()`, which prints each text message.

8-10. Sending Messages: Start with a copy of your program from Exercise 8-9. Write a function called `send_messages()` that prints each text message and moves each message to a new list called `sent_messages` as it's printed. After calling the function, print both of your lists to make sure the messages were moved correctly.

8-11. Archived Messages: Start with your work from Exercise 8-10. Call the function `send_messages()` with a copy of the list of messages. After calling the function, print both of your lists to show that the original list has retained its messages.

Passing an Arbitrary Number of Arguments

Sometimes you won't know ahead of time how many arguments a function needs to accept. Fortunately, Python allows a function to collect an arbitrary number of arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want. The function in the following example has one parameter, `*toppings`, but this parameter collects as many arguments as the calling line provides:

pizza.py

```
def make_pizza(*toppings):
    """Print the list of toppings that have been requested."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the parameter name `*toppings` tells Python to make a tuple called `toppings`, containing all the values this function receives. The `print()` call in the function body produces output showing that Python can handle a function call with one value and a call with three values. It treats the different calls similarly. Note that Python packs the arguments into a tuple, even if the function receives only one value:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Now we can replace the `print()` call with a loop that runs through the list of toppings and describes the pizza being ordered:

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
```

```
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The function responds appropriately, whether it receives one value or three values:

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

This syntax works no matter how many arguments the function receives.

Mixing Positional and Arbitrary Arguments

If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function definition. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter.

For example, if the function needs to take in a size for the pizza, that parameter must come before the parameter `*toppings`:

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

In the function definition, Python assigns the first value it receives to the parameter `size`. All other values that come after are stored in the tuple `toppings`. The function calls include an argument for the size first, followed by as many toppings as needed.

Now each pizza has a size and a number of toppings, and each piece of information is printed in the proper place, showing size first and toppings after:

```
Making a 16-inch pizza with the following toppings:
- pepperoni
```

```
Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

NOTE *You'll often see the generic parameter name `*args`, which collects arbitrary positional arguments like this.*

Using Arbitrary Keyword Arguments

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. One example involves building user profiles: you know you'll get information about a user, but you're not sure what kind of information you'll receive. The function `build_profile()` in the following example always takes in a first and last name, but it accepts an arbitrary number of keyword arguments as well:

user_profile.py

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
❶   user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs as they want. The double asterisks before the parameter `**user_info` cause Python to create a dictionary called `user_info` containing all the extra name-value pairs the function receives. Within the function, you can access the key-value pairs in `user_info` just as you would for any dictionary.

In the body of `build_profile()`, we add the first and last names to the `user_info` dictionary because we'll always receive these two pieces of information from the user ❶, and they haven't been placed into the dictionary yet. Then we return the `user_info` dictionary to the function call line.

We call `build_profile()`, passing it the first name `'albert'`, the last name `'einstein'`, and the two key-value pairs `location='princeton'` and `field='physics'`. We assign the returned `profile` to `user_profile` and print `user_profile`:

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

The returned dictionary contains the user's first and last names and, in this case, the location and field of study as well. The function will work no matter how many additional key-value pairs are provided in the function call.

You can mix positional, keyword, and arbitrary values in many different ways when writing your own functions. It's useful to know that all these argument types exist because you'll see them often when you start reading other people's code. It takes practice to use the different types correctly and to know when to use each type. For now, remember to use the simplest approach that gets the job done. As you progress, you'll learn to use the most efficient approach each time.

NOTE *You'll often see the parameter name* `**kwargs` *used to collect nonspecific keyword arguments.*

TRY IT YOURSELF

8-12. Sandwiches: Write a function that accepts a list of items a person wants on a sandwich. The function should have one parameter that collects as many items as the function call provides, and it should print a summary of the sandwich that's being ordered. Call the function three times, using a different number of arguments each time.

8-13. User Profile: Start with a copy of *user_profile.py* from page 148. Build a profile of yourself by calling `build_profile()`, using your first and last names and three other key-value pairs that describe you.

8-14. Cars: Write a function that stores information about a car in a dictionary. The function should always receive a manufacturer and a model name. It should then accept an arbitrary number of keyword arguments. Call the function with the required information and two other name-value pairs, such as a color or an optional feature. Your function should work for a call like this one:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Print the dictionary that's returned to make sure all the information was stored correctly.

Storing Your Functions in Modules

One advantage of functions is the way they separate blocks of code from your main program. When you use descriptive names for your functions, your programs become much easier to follow. You can go a step further by storing your functions in a separate file called a *module* and then *importing* that module into your main program. An `import` statement tells Python to make the code in a module available in the currently running program file.

Storing your functions in a separate file allows you to hide the details of your program's code and focus on its higher-level logic. It also allows you to reuse functions in many different programs. When you store your functions in separate files, you can share those files with other programmers without

having to share your entire program. Knowing how to import functions also allows you to use libraries of functions that other programmers have written.

There are several ways to import a module, and I'll show you each of these briefly.

Importing an Entire Module

To start importing functions, we first need to create a module. A *module* is a file ending in *.py* that contains the code you want to import into your program. Let's make a module that contains the function `make_pizza()`. To make this module, we'll remove everything from the file *pizza.py* except the function `make_pizza()`:

pizza.py

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

Now we'll make a separate file called *making_pizzas.py* in the same directory as *pizza.py*. This file imports the module we just created and then makes two calls to `make_pizza()`:

making_pizzas.py

```
import pizza

❶ pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

When Python reads this file, the line `import pizza` tells Python to open the file *pizza.py* and copy all the functions from it into this program. You don't actually see code being copied between files because Python copies the code behind the scenes, just before the program runs. All you need to know is that any function defined in *pizza.py* will now be available in *making_pizzas.py*.

To call a function from an imported module, enter the name of the module you imported, `pizza`, followed by the name of the function, `make_pizza()`, separated by a dot ❶. This code produces the same output as the original program that didn't import a module:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

This first approach to importing, in which you simply write `import` followed by the name of the module, makes every function from the module

available in your program. If you use this kind of `import` statement to import an entire module named *module_name.py*, each function in the module is available through the following syntax:

```
module_name.function_name()
```

Importing Specific Functions

You can also import a specific function from a module. Here's the general syntax for this approach:

```
from module_name import function_name
```

You can import as many functions as you want from a module by separating each function's name with a comma:

```
from module_name import function_0, function_1, function_2
```

The *making_pizzas.py* example would look like this if we want to import just the function we're going to use:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

With this syntax, you don't need to use the dot notation when you call a function. Because we've explicitly imported the function `make_pizza()` in the `import` statement, we can call it by name when we use the function.

Using as to Give a Function an Alias

If the name of a function you're importing might conflict with an existing name in your program, or if the function name is long, you can use a short, unique *alias*—an alternate name similar to a nickname for the function. You'll give the function this special nickname when you import the function.

Here we give the function `make_pizza()` an alias, `mp()`, by importing `make_pizza as mp`. The `as` keyword renames a function using the alias you provide:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The `import` statement shown here renames the function `make_pizza()` to `mp()` in this program. Anytime we want to call `make_pizza()` we can simply write `mp()` instead, and Python will run the code in `make_pizza()` while avoiding any confusion with another `make_pizza()` function you might have written in this program file.

The general syntax for providing an alias is:

```
from module_name import function_name as fn
```

Using as to Give a Module an Alias

You can also provide an alias for a module name. Giving a module a short alias, like `p` for `pizza`, allows you to call the module's functions more quickly. Calling `p.make_pizza()` is more concise than calling `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The module `pizza` is given the alias `p` in the `import` statement, but all of the module's functions retain their original names. Calling the functions by writing `p.make_pizza()` is not only more concise than `pizza.make_pizza()`, but it also redirects your attention from the module name and allows you to focus on the descriptive names of its functions. These function names, which clearly tell you what each function does, are more important to the readability of your code than using the full module name.

The general syntax for this approach is:

```
import module_name as mn
```

Importing All Functions in a Module

You can tell Python to import every function in a module by using the asterisk (*) operator:

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the `import` statement tells Python to copy every function from the module `pizza` into this program file. Because every function is imported, you can call each function by name without using the dot notation. However, it's best not to use this approach when you're working with larger modules that you didn't write: if the module has a function name that matches an existing name in your project, you can get unexpected results. Python may see several functions or variables with the same name, and instead of importing all the functions separately, it will overwrite the functions.

The best approach is to import the function or functions you want, or import the entire module and use the dot notation. This leads to clear code that's easy to read and understand. I include this section so you'll recognize `import` statements like the following when you see them in other people's code:

```
from module_name import *
```

Styling Functions

You need to keep a few details in mind when you're styling functions. Functions should have descriptive names, and these names should use lowercase letters and underscores. Descriptive names help you and others understand what your code is trying to do. Module names should use these conventions as well.

Every function should have a comment that explains concisely what the function does. This comment should appear immediately after the function definition and use the docstring format. In a well-documented function, other programmers can use the function by reading only the description in the docstring. They should be able to trust that the code works as described, and as long as they know the name of the function, the arguments it needs, and the kind of value it returns, they should be able to use it in their programs.

If you specify a default value for a parameter, no spaces should be used on either side of the equal sign:

```
def function_name(parameter_0, parameter_1='default value')
```

The same convention should be used for keyword arguments in function calls:

```
function_name(value_0, parameter_1='value')
```

PEP 8 (*https://www.python.org/dev/peps/pep-0008*) recommends that you limit lines of code to 79 characters so every line is visible in a reasonably sized editor window. If a set of parameters causes a function's definition to be longer than 79 characters, press ENTER after the opening parenthesis on the definition line. On the next line, press the TAB key twice to separate the list of arguments from the body of the function, which will only be indented one level.

Most editors automatically line up any additional lines of arguments to match the indentation you have established on the first line:

```
def function_name(
        parameter_0, parameter_1, parameter_2,
        parameter_3, parameter_4, parameter_5):
    function body...
```

If your program or module has more than one function, you can separate each by two blank lines to make it easier to see where one function ends and the next one begins.

All `import` statements should be written at the beginning of a file. The only exception is if you use comments at the beginning of your file to describe the overall program.

TRY IT YOURSELF

8-15. Printing Models: Put the functions for the example *printing_models.py* in a separate file called *printing_functions.py*. Write an import statement at the top of *printing_models.py*, and modify the file to use the imported functions.

8-16. Imports: Using a program you wrote that has one function in it, store that function in a separate file. Import the function into your main program file, and call the function using each of these approaches:

```
import module_name
from module_name import function_name
from module_name import function_name as fn
import module_name as mn
from module_name import *
```

8-17. Styling Functions: Choose any three programs you wrote for this chapter, and make sure they follow the styling guidelines described in this section.

Summary

In this chapter, you learned how to write functions and to pass arguments so that your functions have access to the information they need to do their work. You learned how to use positional and keyword arguments, and also how to accept an arbitrary number of arguments. You saw functions that display output and functions that return values. You learned how to use functions with lists, dictionaries, `if` statements, and `while` loops. You also saw how to store your functions in separate files called *modules*, so your program files will be simpler and easier to understand. Finally, you learned to style your functions so your programs will continue to be well-structured and as easy as possible for you and others to read.

One of your goals as a programmer should be to write simple code that does what you want it to, and functions help you do this. They allow you to write blocks of code and leave them alone once you know they work. When you know a function does its job correctly, you can trust that it will continue to work and move on to your next coding task.

Functions allow you to write code once and then reuse that code as many times as you want. When you need to run the code in a function, all you need to do is write a one-line call and the function does its job. When you need to modify a function's behavior, you only have to modify one block of code, and your change takes effect everywhere you've made a call to that function.

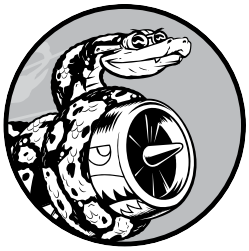
Using functions makes your programs easier to read, and good function names summarize what each part of a program does. Reading a series of function calls gives you a much quicker sense of what a program does than reading a long series of code blocks.

Functions also make your code easier to test and debug. When the bulk of your program's work is done by a set of functions, each of which has a specific job, it's much easier to test and maintain the code you've written. You can write a separate program that calls each function and tests whether each function works in all the situations it may encounter. When you do this, you can be confident that your functions will work properly each time you call them.

In Chapter 9, you'll learn to write classes. *Classes* combine functions and data into one neat package that can be used in flexible and efficient ways.

9

CLASSES



Object-oriented programming (OOP) is one of the most effective approaches to writing software. In object-oriented programming, you write *classes* that represent real-world things and situations, and you create *objects* based on these classes. When you write a class, you define the general behavior that a whole category of objects can have.

When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. You'll be amazed how well real-world situations can be modeled with object-oriented programming.

Making an object from a class is called *instantiation*, and you work with *instances* of a class. In this chapter you'll write classes and create instances of those classes. You'll specify the kind of information that can be stored in instances, and you'll define actions that can be taken with these instances. You'll also write classes that extend the functionality of existing classes, so similar classes can share common functionality, and you can do more with

less code. You'll store your classes in modules and import classes written by other programmers into your own program files.

Learning about object-oriented programming will help you see the world as a programmer does. It'll help you understand your code—not just what's happening line by line, but also the bigger concepts behind it. Knowing the logic behind classes will train you to think logically, so you can write programs that effectively address almost any problem you encounter.

Classes also make life easier for you and the other programmers you'll work with as you take on increasingly complex challenges. When you and other programmers write code based on the same kind of logic, you'll be able to understand each other's work. Your programs will make sense to the people you work with, allowing everyone to accomplish more.

Creating and Using a Class

You can model almost anything using classes. Let's start by writing a simple class, `Dog`, that represents a dog—not one dog in particular, but any dog. What do we know about most pet dogs? Well, they all have a name and an age. We also know that most dogs sit and roll over. Those two pieces of information (name and age) and those two behaviors (sit and roll over) will go in our `Dog` class because they're common to most dogs. This class will tell Python how to make an object representing a dog. After our class is written, we'll use it to make individual instances, each of which represents one specific dog.

Creating the Dog Class

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

dog.py

```
❶ class Dog:
      """A simple attempt to model a dog."""

❷     def __init__(self, name, age):
          """Initialize name and age attributes."""
❸         self.name = name
          self.age = age

❹     def sit(self):
          """Simulate a dog sitting in response to a command."""
          print(f"{self.name} is now sitting.")

      def roll_over(self):
          """Simulate rolling over in response to a command."""
          print(f"{self.name} rolled over!")
```

There's a lot to notice here, but don't worry. You'll see this structure throughout this chapter and have lots of time to get used to it. We first define a class called `Dog` ❶. By convention, capitalized names refer to classes in Python. There are no parentheses in the class definition because we're creating this class from scratch. We then write a docstring describing what this class does.

The __init__() Method

A function that's part of a class is a *method*. Everything you learned about functions applies to methods as well; the only practical difference for now is the way we'll call methods. The `__init__()` method ❷ is a special method that Python runs automatically whenever we create a new instance based on the `Dog` class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names. Make sure to use two underscores on each side of `__init__()`. If you use just one on each side, the method won't be called automatically when you use your class, which can result in errors that are difficult to identify.

We define the `__init__()` method to have three parameters: `self`, `name`, and `age`. The `self` parameter is required in the method definition, and it must come first, before the other parameters. It must be included in the definition because when Python calls this method later (to create an instance of `Dog`), the method call will automatically pass the `self` argument. Every method call associated with an instance automatically passes `self`, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of `Dog`, Python will call the `__init__()` method from the `Dog` class. We'll pass `Dog()` a name and an age as arguments; `self` is passed automatically, so we don't need to pass it. Whenever we want to make an instance from the `Dog` class, we'll provide values for only the last two parameters, `name` and `age`.

The two variables defined in the body of the `__init__()` method each have the prefix `self` ❸. Any variable prefixed with `self` is available to every method in the class, and we'll also be able to access these variables through any instance created from the class. The line `self.name = name` takes the value associated with the parameter `name` and assigns it to the variable `name`, which is then attached to the instance being created. The same process happens with `self.age = age`. Variables that are accessible through instances like this are called *attributes*.

The `Dog` class has two other methods defined: `sit()` and `roll_over()` ❹. Because these methods don't need additional information to run, we just define them to have one parameter, `self`. The instances we create later will have access to these methods. In other words, they'll be able to sit and roll over. For now, `sit()` and `roll_over()` don't do much. They simply print a message saying the dog is sitting or rolling over. But the concept can be extended to realistic situations: if this class were part of a computer game, these methods would contain code to make an animated dog sit and roll over. If this class was written to control a robot, these methods would direct movements that cause a robotic dog to sit and roll over.

Making an Instance from a Class

Think of a class as a set of instructions for how to make an instance. The `Dog` class is a set of instructions that tells Python how to make individual instances representing specific dogs.

Let's make an instance representing a specific dog:

```
class Dog:
    --snip--

❶ my_dog = Dog('Willie', 6)

❷ print(f"My dog's name is {my_dog.name}.")
❸ print(f"My dog is {my_dog.age} years old.")
```

The `Dog` class we're using here is the one we just wrote in the previous example. Here, we tell Python to create a dog whose name is `'Willie'` and whose age is `6` ❶. When Python reads this line, it calls the `__init__()` method in `Dog` with the arguments `'Willie'` and `6`. The `__init__()` method creates an instance representing this particular dog and sets the `name` and `age` attributes using the values we provided. Python then returns an instance representing this dog. We assign that instance to the variable `my_dog`. The naming convention is helpful here; we can usually assume that a capitalized name like `Dog` refers to a class, and a lowercase name like `my_dog` refers to a single instance created from a class.

Accessing Attributes

To access the attributes of an instance, you use dot notation. We access the value of `my_dog`'s attribute `name` ❷ by writing:

```
my_dog.name
```

Dot notation is used often in Python. This syntax demonstrates how Python finds an attribute's value. Here, Python looks at the instance `my_dog` and then finds the attribute `name` associated with `my_dog`. This is the same attribute referred to as `self.name` in the class `Dog`. We use the same approach to work with the attribute `age` ❸.

The output is a summary of what we know about `my_dog`:

```
My dog's name is Willie.
My dog is 6 years old.
```

Calling Methods

After we create an instance from the class `Dog`, we can use dot notation to call any method defined in `Dog`. Let's make our dog sit and roll over:

```
class Dog:
    --snip--

my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

To call a method, give the name of the instance (in this case, `my_dog`) and the method you want to call, separated by a dot. When Python reads `my_dog.sit()`, it looks for the method `sit()` in the class `Dog` and runs that code. Python interprets the line `my_dog.roll_over()` in the same way.

Now Willie does what we tell him to:

```
Willie is now sitting.
Willie rolled over!
```

This syntax is quite useful. When attributes and methods have been given appropriately descriptive names like `name`, `age`, `sit()`, and `roll_over()`, we can easily infer what a block of code, even one we've never seen before, is supposed to do.

Creating Multiple Instances

You can create as many instances from a class as you need. Let's create a second dog called `your_dog`:

```
class Dog:
    --snip--

my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

In this example we create a dog named Willie and a dog named Lucy. Each dog is a separate instance with its own set of attributes, capable of the same set of actions:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Even if we used the same name and age for the second dog, Python would still create a separate instance from the `Dog` class. You can make as many instances from one class as you need, as long as you give each instance a unique variable name or it occupies a unique spot in a list or dictionary.

TRY IT YOURSELF

9-1. Restaurant: Make a class called `Restaurant`. The `__init__()` method for `Restaurant` should store two attributes: a `restaurant_name` and a `cuisine_type`. Make a method called `describe_restaurant()` that prints these two pieces of information, and a method called `open_restaurant()` that prints a message indicating that the restaurant is open.

Make an instance called `restaurant` from your class. Print the two attributes individually, and then call both methods.

9-2. Three Restaurants: Start with your class from Exercise 9-1. Create three different instances from the class, and call `describe_restaurant()` for each instance.

9-3. Users: Make a class called `User`. Create two attributes called `first_name` and `last_name`, and then create several other attributes that are typically stored in a user profile. Make a method called `describe_user()` that prints a summary of the user's information. Make another method called `greet_user()` that prints a personalized greeting to the user.

Create several instances representing different users, and call both methods for each user.

Working with Classes and Instances

You can use classes to represent many real-world situations. Once you write a class, you'll spend most of your time working with instances created from that class. One of the first tasks you'll want to do is modify the attributes associated with a particular instance. You can modify the attributes of an instance directly or write methods that update attributes in specific ways.

The Car Class

Let's write a new class representing a car. Our class will store information about the kind of car we're working with, and it will have a method that summarizes this information:

car.py

```
class Car:
    """A simple attempt to represent a car."""

❶   def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year

❷   def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
```

```
        return long_name.title()

❸ my_new_car = Car('audi', 'a4', 2024)
  print(my_new_car.get_descriptive_name())
```

In the `Car` class, we define the `__init__()` method with the `self` parameter first ❶, just like we did with the `Dog` class. We also give it three other parameters: `make`, `model`, and `year`. The `__init__()` method takes in these parameters and assigns them to the attributes that will be associated with instances made from this class. When we make a new `Car` instance, we'll need to specify a make, model, and year for our instance.

We define a method called `get_descriptive_name()` ❷ that puts a car's `year`, `make`, and `model` into one string neatly describing the car. This will spare us from having to print each attribute's value individually. To work with the attribute values in this method, we use `self.make`, `self.model`, and `self.year`. Outside of the class, we make an instance from the `Car` class and assign it to the variable `my_new_car` ❸. Then we call `get_descriptive_name()` to show what kind of car we have:

```
2024 Audi A4
```

To make the class more interesting, let's add an attribute that changes over time. We'll add an attribute that stores the car's overall mileage.

Setting a Default Value for an Attribute

When an instance is created, attributes can be defined without being passed in as parameters. These attributes can be defined in the `__init__()` method, where they are assigned a default value.

Let's add an attribute called `odometer_reading` that always starts with a value of 0. We'll also add a method `read_odometer()` that helps us read each car's odometer:

```
class Car:

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
❶       self.odometer_reading = 0

    def get_descriptive_name(self):
        --snip--

❷   def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

This time, when Python calls the `__init__()` method to create a new instance, it stores the make, model, and year values as attributes, like it did in the previous example. Then Python creates a new attribute called `odometer_reading` and sets its initial value to 0 ❶. We also have a new method called `read_odometer()` ❷ that makes it easy to read a car's mileage.

Our car starts with a mileage of 0:

```
2024 Audi A4
This car has 0 miles on it.
```

Not many cars are sold with exactly 0 miles on the odometer, so we need a way to change the value of this attribute.

Modifying Attribute Values

You can change an attribute's value in three ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method. Let's look at each of these approaches.

Modifying an Attribute's Value Directly

The simplest way to modify the value of an attribute is to access the attribute directly through an instance. Here we set the odometer reading to 23 directly:

```
class Car:
    --snip--

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

We use dot notation to access the car's `odometer_reading` attribute, and set its value directly. This line tells Python to take the instance `my_new_car`, find the attribute `odometer_reading` associated with it, and set the value of that attribute to 23:

```
2024 Audi A4
This car has 23 miles on it.
```

Sometimes you'll want to access attributes directly like this, but other times you'll want to write a method that updates the value for you.

Modifying an Attribute's Value Through a Method

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

Here's an example showing a method called `update_odometer()`:

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

❶ my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

The only modification to `Car` is the addition of `update_odometer()`. This method takes in a mileage value and assigns it to `self.odometer_reading`. Using the `my_new_car` instance, we call `update_odometer()` with `23` as an argument ❶. This sets the odometer reading to 23, and `read_odometer()` prints the reading:

```
2024 Audi A4
This car has 23 miles on it.
```

We can extend the method `update_odometer()` to do additional work every time the odometer reading is modified. Let's add a little logic to make sure no one tries to roll back the odometer reading:

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
❶       if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
❷           print("You can't roll back an odometer!")
```

Now `update_odometer()` checks that the new reading makes sense before modifying the attribute. If the value provided for `mileage` is greater than or equal to the existing mileage, `self.odometer_reading`, you can update the odometer reading to the new mileage ❶. If the new mileage is less than the existing mileage, you'll get a warning that you can't roll back an odometer ❷.

Incrementing an Attribute's Value Through a Method

Sometimes you'll want to increment an attribute's value by a certain amount, rather than set an entirely new value. Say we buy a used car and put 100 miles

on it between the time we buy it and the time we register it. Here's a method that allows us to pass this incremental amount and add that value to the odometer reading:

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        --snip--

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

❶ my_used_car = Car('subaru', 'outback', 2019)
print(my_used_car.get_descriptive_name())

❷ my_used_car.update_odometer(23_500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

The new method `increment_odometer()` takes in a number of miles, and adds this value to `self.odometer_reading`. First, we create a used car, `my_used_car` ❶. We set its odometer to 23,500 by calling `update_odometer()` and passing it `23_500` ❷. Finally, we call `increment_odometer()` and pass it `100` to add the 100 miles that we drove between buying the car and registering it:

```
2019 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

You can modify this method to reject negative increments so no one uses this function to roll back an odometer as well.

NOTE *You can use methods like this to control how users of your program update values such as an odometer reading, but anyone with access to the program can set the odometer reading to any value by accessing the attribute directly. Effective security takes extreme attention to detail in addition to basic checks like those shown here.*

TRY IT YOURSELF

9-4. Number Served: Start with your program from Exercise 9-1 (page 162). Add an attribute called `number_served` with a default value of 0. Create an instance called `restaurant` from this class. Print the number of customers the restaurant has served, and then change this value and print it again.

Add a method called `set_number_served()` that lets you set the number of customers that have been served. Call this method with a new number and print the value again.

Add a method called `increment_number_served()` that lets you increment the number of customers who've been served. Call this method with any number you like that could represent how many customers were served in, say, a day of business.

9-5. Login Attempts: Add an attribute called `login_attempts` to your `User` class from Exercise 9-3 (page 162). Write a method called `increment_login_attempts()` that increments the value of `login_attempts` by 1. Write another method called `reset_login_attempts()` that resets the value of `login_attempts` to 0.

Make an instance of the `User` class and call `increment_login_attempts()` several times. Print the value of `login_attempts` to make sure it was incremented properly, and then call `reset_login_attempts()`. Print `login_attempts` again to make sure it was reset to 0.

Inheritance

You don't always have to start from scratch when writing a class. If the class you're writing is a specialized version of another class you wrote, you can use *inheritance*. When one class *inherits* from another, it takes on the attributes and methods of the first class. The original class is called the *parent class*, and the new class is the *child class*. The child class can inherit any or all of the attributes and methods of its parent class, but it's also free to define new attributes and methods of its own.

The __init__() Method for a Child Class

When you're writing a new class based on an existing class, you'll often want to call the `__init__()` method from the parent class. This will initialize any attributes that were defined in the parent `__init__()` method and make them available in the child class.

As an example, let's model an electric car. An electric car is just a specific kind of car, so we can base our new `ElectricCar` class on the `Car` class we wrote earlier. Then we'll only have to write code for the attributes and behaviors specific to electric cars.

Let's start by making a simple version of the `ElectricCar` class, which does everything the `Car` class does:

electric_car.py

```
❶ class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
```

```
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

❷ class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

❸     def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
❹         super().__init__(make, model, year)


❺ my_leaf = ElectricCar('nissan', 'leaf', 2024)
  print(my_leaf.get_descriptive_name())
```

We start with `Car` ❶. When you create a child class, the parent class must be part of the current file and must appear before the child class in the file. We then define the child class, `ElectricCar` ❷. The name of the parent class must be included in parentheses in the definition of a child class. The `__init__()` method takes in the information required to make a `Car` instance ❸.

The `super()` function ❹ is a special function that allows you to call a method from the parent class. This line tells Python to call the `__init__()` method from `Car`, which gives an `ElectricCar` instance all the attributes defined in that method. The name *super* comes from a convention of calling the parent class a *superclass* and the child class a *subclass*.

We test whether inheritance is working properly by trying to create an electric car with the same kind of information we'd provide when making a regular car. We make an instance of the `ElectricCar` class and assign it to `my_leaf` ❺. This line calls the `__init__()` method defined in `ElectricCar`, which in turn tells Python to call the `__init__()` method defined in the parent class `Car`. We provide the arguments `'nissan'`, `'leaf'`, and `2024`.

Aside from `__init__()`, there are no attributes or methods yet that are particular to an electric car. At this point we're just making sure the electric car has the appropriate `Car` behaviors:

```
2024 Nissan Leaf
```

The `ElectricCar` instance works just like an instance of `Car`, so now we can begin defining attributes and methods specific to electric cars.

Defining Attributes and Methods for the Child Class

Once you have a child class that inherits from a parent class, you can add any new attributes and methods necessary to differentiate the child class from the parent class.

Let's add an attribute that's specific to electric cars (a battery, for example) and a method to report on this attribute. We'll store the battery size and write a method that prints a description of the battery:

```
class Car:
    --snip--

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
❶       self.battery_size = 40

❷   def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.describe_battery()
```

We add a new attribute `self.battery_size` and set its initial value to `40` ❶. This attribute will be associated with all instances created from the `ElectricCar` class but won't be associated with any instances of `Car`. We also add a method called `describe_battery()` that prints information about the battery ❷. When we call this method, we get a description that is clearly specific to an electric car:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
```

There's no limit to how much you can specialize the `ElectricCar` class. You can add as many attributes and methods as you need to model an

electric car to whatever degree of accuracy you need. An attribute or method that could belong to any car, rather than one that's specific to an electric car, should be added to the `Car` class instead of the `ElectricCar` class. Then anyone who uses the `Car` class will have that functionality available as well, and the `ElectricCar` class will only contain code for the information and behavior specific to electric vehicles.

Overriding Methods from the Parent Class

You can override any method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class with the same name as the method you want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

Say the class `Car` had a method called `fill_gas_tank()`. This method is meaningless for an all-electric vehicle, so you might want to override this method. Here's one way to do that:

```
class ElectricCar(Car):
    --snip--

    def fill_gas_tank(self):
        """Electric cars don't have gas tanks."""
        print("This car doesn't have a gas tank!")
```

Now if someone tries to call `fill_gas_tank()` with an electric car, Python will ignore the method `fill_gas_tank()` in `Car` and run this code instead. When you use inheritance, you can make your child classes retain what you need and override anything you don't need from the parent class.

Instances as Attributes

When modeling something from the real world in code, you may find that you're adding more and more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy. In these situations, you might recognize that part of one class can be written as a separate class. You can break your large class into smaller classes that work together; this approach is called *composition*.

For example, if we continue adding detail to the `ElectricCar` class, we might notice that we're adding many attributes and methods specific to the car's battery. When we see this happening, we can stop and move those attributes and methods to a separate class called `Battery`. Then we can use a `Battery` instance as an attribute in the `ElectricCar` class:

```
class Car:
    --snip--

class Battery:
    """A simple attempt to model a battery for an electric car."""

❶   def __init__(self, battery_size=40):
```

```
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

❷   def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")


class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
❸       self.battery = Battery()

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
```

We define a new class called `Battery` that doesn't inherit from any other class. The `__init__()` method ❶ has one parameter, `battery_size`, in addition to `self`. This is an optional parameter that sets the battery's size to 40 if no value is provided. The method `describe_battery()` has been moved to this class as well ❷.

In the `ElectricCar` class, we now add an attribute called `self.battery` ❸. This line tells Python to create a new instance of `Battery` (with a default size of 40, because we're not specifying a value) and assign that instance to the attribute `self.battery`. This will happen every time the `__init__()` method is called; any `ElectricCar` instance will now have a `Battery` instance created automatically.

We create an electric car and assign it to the variable `my_leaf`. When we want to describe the battery, we need to work through the car's `battery` attribute:

```
my_leaf.battery.describe_battery()
```

This line tells Python to look at the instance `my_leaf`, find its `battery` attribute, and call the method `describe_battery()` that's associated with the `Battery` instance assigned to the attribute.

The output is identical to what we saw previously:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
```

This looks like a lot of extra work, but now we can describe the battery in as much detail as we want without cluttering the `ElectricCar` class. Let's

add another method to Battery that reports the range of the car based on the battery size:

```
class Car:
    --snip--

class Battery:
    --snip--

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 40:
            range = 150
        elif self.battery_size == 65:
            range = 225

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    --snip--

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
❶ my_leaf.battery.get_range()
```

The new method get_range() performs some simple analysis. If the battery's capacity is 40 kWh, get_range() sets the range to 150 miles, and if the capacity is 65 kWh, it sets the range to 225 miles. It then reports this value. When we want to use this method, we again have to call it through the car's battery attribute ❶.

The output tells us the range of the car based on its battery size:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
This car can go about 150 miles on a full charge.
```

Modeling Real-World Objects

As you begin to model more complicated things like electric cars, you'll wrestle with interesting questions. Is the range of an electric car a property of the battery or of the car? If we're only describing one car, it's probably fine to maintain the association of the method get_range() with the Battery class. But if we're describing a manufacturer's entire line of cars, we probably want to move get_range() to the ElectricCar class. The get_range() method would still check the battery size before determining the range, but it would report a range specific to the kind of car it's associated with. Alternatively, we could maintain the association of the get_range() method with the battery but pass it a parameter such as car_model. The get_range() method would then report a range based on the battery size and car model.

This brings you to an interesting point in your growth as a programmer. When you wrestle with questions like these, you're thinking at a higher

logical level rather than a syntax-focused level. You're thinking not about Python, but about how to represent the real world in code. When you reach this point, you'll realize there are often no right or wrong approaches to modeling real-world situations. Some approaches are more efficient than others, but it takes practice to find the most efficient representations. If your code is working as you want it to, you're doing well! Don't be discouraged if you find you're ripping apart your classes and rewriting them several times using different approaches. In the quest to write accurate, efficient code, everyone goes through this process.

TRY IT YOURSELF

9-6. Ice Cream Stand: An ice cream stand is a specific kind of restaurant. Write a class called `IceCreamStand` that inherits from the `Restaurant` class you wrote in Exercise 9-1 (page 162) or Exercise 9-4 (page 166). Either version of the class will work; just pick the one you like better. Add an attribute called `flavors` that stores a list of ice cream flavors. Write a method that displays these flavors. Create an instance of `IceCreamStand`, and call this method.

9-7. Admin: An administrator is a special kind of user. Write a class called `Admin` that inherits from the `User` class you wrote in Exercise 9-3 (page 162) or Exercise 9-5 (page 167). Add an attribute, `privileges`, that stores a list of strings like `"can add post"`, `"can delete post"`, `"can ban user"`, and so on. Write a method called `show_privileges()` that lists the administrator's set of privileges. Create an instance of `Admin`, and call your method.

9-8. Privileges: Write a separate `Privileges` class. The class should have one attribute, `privileges`, that stores a list of strings as described in Exercise 9-7. Move the `show_privileges()` method to this class. Make a `Privileges` instance as an attribute in the `Admin` class. Create a new instance of `Admin` and use your method to show its privileges.

9-9. Battery Upgrade: Use the final version of *electric_car.py* from this section. Add a method to the `Battery` class called `upgrade_battery()`. This method should check the battery size and set the capacity to 65 if it isn't already. Make an electric car with a default battery size, call `get_range()` once, and then call `get_range()` a second time after upgrading the battery. You should see an increase in the car's range.

Importing Classes

As you add more functionality to your classes, your files can get long, even when you use inheritance and composition properly. In keeping with the overall philosophy of Python, you'll want to keep your files as uncluttered as possible. To help, Python lets you store classes in modules and then import the classes you need into your main program.

Importing a Single Class

Let's create a module containing just the `Car` class. This brings up a subtle naming issue: we already have a file named *car.py* in this chapter, but this module should be named *car.py* because it contains code representing a car. We'll resolve this naming issue by storing the `Car` class in a module named *car.py*, replacing the *car.py* file we were previously using. From now on, any program that uses this module will need a more specific filename, such as *my_car.py*. Here's *car.py* with just the code from the class `Car`:

car.py

```
❶ """A class that can be used to represent a car."""

class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles
```

We include a module-level docstring that briefly describes the contents of this module ❶. You should write a docstring for each module you create.

Now we make a separate file called *my_car.py*. This file will import the `Car` class and then create an instance from that class:

my_car.py

```
❶ from car import Car

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

```
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

The `import` statement ❶ tells Python to open the `car` module and import the class `Car`. Now we can use the `Car` class as if it were defined in this file. The output is the same as we saw earlier:

```
2024 Audi A4
This car has 23 miles on it.
```

Importing classes is an effective way to program. Picture how long this program file would be if the entire `Car` class were included. When you instead move the class to a module and import the module, you still get all the same functionality, but you keep your main program file clean and easy to read. You also store most of the logic in separate files; once your classes work as you want them to, you can leave those files alone and focus on the higher-level logic of your main program.

Storing Multiple Classes in a Module

You can store as many classes as you need in a single module, although each class in a module should be related somehow. The classes `Battery` and `ElectricCar` both help represent cars, so let's add them to the module *car.py*.

car.py

```
"""A set of classes used to represent gas and electric cars."""

class Car:
    --snip--

class Battery:
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=40):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 40:
            range = 150
        elif self.battery_size == 65:
            range = 225

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Models aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
```

```
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

Now we can make a new file called *my_electric_car.py*, import the `ElectricCar` class, and make an electric car:

my_electric_car.py

```
from car import ElectricCar

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
my_leaf.battery.get_range()
```

This has the same output we saw earlier, even though most of the logic is hidden away in a module:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
This car can go about 150 miles on a full charge.
```

Importing Multiple Classes from a Module

You can import as many classes as you need into a program file. If we want to make a regular car and an electric car in the same file, we need to import both classes, `Car` and `ElectricCar`:

my_cars.py

```
❶ from car import Car, ElectricCar

❷ my_mustang = Car('ford', 'mustang', 2024)
  print(my_mustang.get_descriptive_name())
❸ my_leaf = ElectricCar('nissan', 'leaf', 2024)
  print(my_leaf.get_descriptive_name())
```

You import multiple classes from a module by separating each class with a comma ❶. Once you've imported the necessary classes, you're free to make as many instances of each class as you need.

In this example we make a gas-powered Ford Mustang ❷ and then an electric Nissan Leaf ❸:

```
2024 Ford Mustang
2024 Nissan Leaf
```

Importing an Entire Module

You can also import an entire module and then access the classes you need using dot notation. This approach is simple and results in code that is easy to read. Because every call that creates an instance of a class includes the module name, you won't have naming conflicts with any names used in the current file.

Here's what it looks like to import the entire `car` module and then create a regular car and an electric car:

my_cars.py

```
❶ import car

❷ my_mustang = car.Car('ford', 'mustang', 2024)
  print(my_mustang.get_descriptive_name())

❸ my_leaf = car.ElectricCar('nissan', 'leaf', 2024)
  print(my_leaf.get_descriptive_name())
```

First we import the entire `car` module ❶. We then access the classes we need through the *`module_name.ClassName`* syntax. We again create a Ford Mustang ❷, and a Nissan Leaf ❸.

Importing All Classes from a Module

You can import every class from a module using the following syntax:

```
from module_name import *
```

This method is not recommended for two reasons. First, it's helpful to be able to read the `import` statements at the top of a file and get a clear sense of which classes a program uses. With this approach it's unclear which classes you're using from the module. This approach can also lead to confusion with names in the file. If you accidentally import a class with the same name as something else in your program file, you can create errors that are hard to diagnose. I show this here because even though it's not a recommended approach, you're likely to see it in other people's code at some point.

If you need to import many classes from a module, you're better off importing the entire module and using the *`module_name.ClassName`* syntax. You won't see all the classes used at the top of the file, but you'll see clearly where the module is used in the program. You'll also avoid the potential naming conflicts that can arise when you import every class in a module.

Importing a Module into a Module

Sometimes you'll want to spread out your classes over several modules to keep any one file from growing too large and avoid storing unrelated classes in the same module. When you store your classes in several modules, you may find that a class in one module depends on a class in another module. When this happens, you can import the required class into the first module.

For example, let's store the `Car` class in one module and the `ElectricCar` and `Battery` classes in a separate module. We'll make a new module called *electric_car.py*—replacing the *electric_car.py* file we created earlier—and copy just the `Battery` and `ElectricCar` classes into this file:

electric_car.py

```
"""A set of classes that can be used to represent electric cars."""

from car import Car
```

```
class Battery:
    --snip--

class ElectricCar(Car):
    --snip--
```

The class `ElectricCar` needs access to its parent class `Car`, so we import `Car` directly into the module. If we forget this line, Python will raise an error when we try to import the `electric_car` module. We also need to update the `Car` module so it contains only the `Car` class:

car.py

```
"""A class that can be used to represent a car."""

class Car:
    --snip--
```

Now we can import from each module separately and create whatever kind of car we need:

my_cars.py

```
from car import Car
from electric_car import ElectricCar

my_mustang = Car('ford', 'mustang', 2024)
print(my_mustang.get_descriptive_name())

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

We import `Car` from its module, and `ElectricCar` from its module. We then create one regular car and one electric car. Both cars are created correctly:

```
2024 Ford Mustang
2024 Nissan Leaf
```

Using Aliases

As you saw in Chapter 8, aliases can be quite helpful when using modules to organize your projects' code. You can use aliases when importing classes as well.

As an example, consider a program where you want to make a bunch of electric cars. It might get tedious to type (and read) `ElectricCar` over and over again. You can give `ElectricCar` an alias in the import statement:

```
from electric_car import ElectricCar as EC
```

Now you can use this alias whenever you want to make an electric car:

```
my_leaf = EC('nissan', 'leaf', 2024)
```

You can also give a module an alias. Here's how to import the entire `electric_car` module using an alias:

```
import electric_car as ec
```

Now you can use this module alias with the full class name:

```
my_leaf = ec.ElectricCar('nissan', 'leaf', 2024)
```

Finding Your Own Workflow

As you can see, Python gives you many options for how to structure code in a large project. It's important to know all these possibilities so you can determine the best ways to organize your projects as well as understand other people's projects.

When you're starting out, keep your code structure simple. Try doing everything in one file and moving your classes to separate modules once everything is working. If you like how modules and files interact, try storing your classes in modules when you start a project. Find an approach that lets you write code that works, and go from there.

TRY IT YOURSELF

9-10. Imported Restaurant: Using your latest `Restaurant` class, store it in a module. Make a separate file that imports `Restaurant`. Make a `Restaurant` instance, and call one of `Restaurant`'s methods to show that the `import` statement is working properly.

9-11. Imported Admin: Start with your work from Exercise 9-8 (page 173). Store the classes `User`, `Privileges`, and `Admin` in one module. Create a separate file, make an `Admin` instance, and call `show_privileges()` to show that everything is working correctly.

9-12. Multiple Modules: Store the `User` class in one module, and store the `Privileges` and `Admin` classes in a separate module. In a separate file, create an `Admin` instance and call `show_privileges()` to show that everything is still working correctly.

The Python Standard Library

The *Python standard library* is a set of modules included with every Python installation. Now that you have a basic understanding of how functions and classes work, you can start to use modules like these that other programmers have written. You can use any function or class in the standard library by including a simple `import` statement at the top of your file. Let's look at one module, `random`, which can be useful in modeling many real-world situations.

One interesting function from the random module is `randint()`. This function takes two integer arguments and returns a randomly selected integer between (and including) those numbers.

Here's how to generate a random number between 1 and 6:

```
>>> from random import randint
>>> randint(1, 6)
3
```

Another useful function is `choice()`. This function takes in a list or tuple and returns a randomly chosen element:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

The `random` module shouldn't be used when building security-related applications, but it works well for many fun and interesting projects.

NOTE *You can also download modules from external sources. You'll see a number of these examples in Part II, where we'll need external modules to complete each project.*

TRY IT YOURSELF

9-13. Dice: Make a class `Die` with one attribute called `sides`, which has a default value of 6. Write a method called `roll_die()` that prints a random number between 1 and the number of sides the die has. Make a 6-sided die and roll it 10 times.

Make a 10-sided die and a 20-sided die. Roll each die 10 times.

9-14. Lottery: Make a list or tuple containing a series of 10 numbers and 5 letters. Randomly select 4 numbers or letters from the list and print a message saying that any ticket matching these 4 numbers or letters wins a prize.

9-15. Lottery Analysis: You can use a loop to see how hard it might be to win the kind of lottery you just modeled. Make a list or tuple called `my_ticket`. Write a loop that keeps pulling numbers until your ticket wins. Print a message reporting how many times the loop had to run to give you a winning ticket.

9-16. Python Module of the Week: One excellent resource for exploring the Python standard library is a site called *Python Module of the Week*. Go to *https://pymotw.com* and look at the table of contents. Find a module that looks interesting to you and read about it, perhaps starting with the `random` module.

Styling Classes

A few styling issues related to classes are worth clarifying, especially as your programs become more complicated.

Class names should be written in *CamelCase*. To do this, capitalize the first letter of each word in the name, and don't use underscores. Instance and module names should be written in lowercase, with underscores between words.

Every class should have a docstring immediately following the class definition. The docstring should be a brief description of what the class does, and you should follow the same formatting conventions you used for writing docstrings in functions. Each module should also have a docstring describing what the classes in a module can be used for.

You can use blank lines to organize code, but don't use them excessively. Within a class you can use one blank line between methods, and within a module you can use two blank lines to separate classes.

If you need to import a module from the standard library and a module that you wrote, place the import statement for the standard library module first. Then add a blank line and the import statement for the module you wrote. In programs with multiple import statements, this convention makes it easier to see where the different modules used in the program come from.

Summary

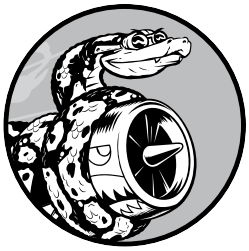
In this chapter, you learned how to write your own classes. You learned how to store information in a class using attributes and how to write methods that give your classes the behavior they need. You learned to write `__init__()` methods that create instances from your classes with exactly the attributes you want. You saw how to modify the attributes of an instance directly and through methods. You learned that inheritance can simplify the creation of classes that are related to each other, and you learned to use instances of one class as attributes in another class to keep each class simple.

You saw how storing classes in modules and importing classes you need into the files where they'll be used can keep your projects organized. You started learning about the Python standard library, and you saw an example based on the `random` module. Finally, you learned to style your classes using Python conventions.

In Chapter 10, you'll learn to work with files so you can save the work you've done in a program and the work you've allowed users to do. You'll also learn about *exceptions*, a special Python class designed to help you respond to errors when they arise.

10

FILES AND EXCEPTIONS



Now that you've mastered the basic skills you need to write organized programs that are easy to use, it's time to think about making your programs even more relevant and usable. In this chapter, you'll learn to work with files so your programs can quickly analyze lots of data.

You'll learn to handle errors so your programs don't crash when they encounter unexpected situations. You'll learn about *exceptions*, which are special objects Python creates to manage errors that arise while a program is running. You'll also learn about the `json` module, which allows you to save user data so it isn't lost when your program stops running.

Learning to work with files and save data will make your programs easier for people to use. Users will be able to choose what data to enter and when to enter it. People will be able to run your program, do some work, and then close the program and pick up where they left off. Learning to handle exceptions will help you deal with situations in which files don't exist and deal with other problems that can cause your programs to crash. This will make your programs more robust when they encounter bad data, whether it comes from

innocent mistakes or from malicious attempts to break your programs. With the skills you'll learn in this chapter, you'll make your programs more applicable, usable, and stable.

Reading from a File

An incredible amount of data is available in text files. Text files can contain weather data, traffic data, socioeconomic data, literary works, and more. Reading from a file is particularly useful in data analysis applications, but it's also applicable to any situation in which you want to analyze or modify information stored in a file. For example, you can write a program that reads in the contents of a text file and rewrites the file with formatting that allows a browser to display it.

When you want to work with the information in a text file, the first step is to read the file into memory. You can then work through all of the file's contents at once or work through the contents line by line.

Reading the Contents of a File

To begin, we need a file with a few lines of text in it. Let's start with a file that contains *pi* to 30 decimal places, with 10 decimal places per line:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

To try the following examples yourself, you can enter these lines in an editor and save the file as *pi_digits.txt*, or you can download the file from the book's resources through *https://ehmatthes.github.io/pcc_3e*. Save the file in the same directory where you'll store this chapter's programs.

Here's a program that opens this file, reads it, and prints the contents of the file to the screen:

file_reader.py

```
from pathlib import Path

❶ path = Path('pi_digits.txt')
❷ contents = path.read_text()
print(contents)
```

To work with the contents of a file, we need to tell Python the path to the file. A *path* is the exact location of a file or folder on a system. Python provides a module called `pathlib` that makes it easier to work with files and directories, no matter which operating system you or your program's users are working with. A module that provides specific functionality like this is often called a *library*, hence the name `pathlib`.

We start by importing the `Path` class from `pathlib`. There's a lot you can do with a `Path` object that points to a file. For example, you can check that the file exists before working with it, read the file's contents, or write new data to the file. Here, we build a `Path` object representing the file *pi_digits.txt*, which we assign to the variable `path` ❶. Since this file is saved in the same

directory as the *.py* file we're writing, the filename is all that `Path` needs to access the file.

NOTE *VS Code looks for files in the folder that was most recently opened. If you're using VS Code, start by opening the folder where you're storing this chapter's programs. For example, if you're saving your program files in a folder called* chapter_10*, press CTRL-O (⌘-O on macOS), and open that folder.*

Once we have a `Path` object representing *pi_digits.txt*, we use the `read_text()` method to read the entire contents of the file ❷. The contents of the file are returned as a single string, which we assign to the variable `contents`. When we print the value of `contents`, we see the entire contents of the text file:

```
3.1415926535
  8979323846
  2643383279

```

The only difference between this output and the original file is the extra blank line at the end of the output. The blank line appears because `read_text()` returns an empty string when it reaches the end of the file; this empty string shows up as a blank line.

We can remove the extra blank line by using `rstrip()` on the `contents` string:

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
contents = contents.rstrip()
print(contents)
```

Recall from Chapter 2 that Python's `rstrip()` method removes, or strips, any whitespace characters from the right side of a string. Now the output matches the contents of the original file exactly:

```
3.1415926535
  8979323846
  2643383279
```

We can strip the trailing newline character when we read the contents of the file, by applying the `rstrip()` method immediately after calling `read_text()`:

```
contents = path.read_text().rstrip()
```

This line tells Python to call the `read_text()` method on the file we're working with. Then it applies the `rstrip()` method to the string that `read_text()` returns. The cleaned-up string is then assigned to the variable `contents`. This approach is called *method chaining*, and you'll see it used often in programming.

Relative and Absolute File Paths

When you pass a simple filename like *pi_digits.txt* to `Path`, Python looks in the directory where the file that's currently being executed (that is, your *.py* program file) is stored.

Sometimes, depending on how you organize your work, the file you want to open won't be in the same directory as your program file. For example, you might store your program files in a folder called *python_work*; inside *python_work*, you might have another folder called *text_files* to distinguish your program files from the text files they're manipulating. Even though *text_files* is in *python_work*, just passing `Path` the name of a file in *text_files* won't work, because Python will only look in *python_work* and stop there; it won't go on and look in *text_files*. To get Python to open files from a directory other than the one where your program file is stored, you need to provide the correct path.

There are two main ways to specify paths in programming. A *relative file path* tells Python to look for a given location relative to the directory where the currently running program file is stored. Since *text_files* is inside *python_work*, we need to build a path that starts with the directory *text_files*, and ends with the filename. Here's how to build this path:

```
path = Path('text_files/filename.txt')
```

You can also tell Python exactly where the file is on your computer, regardless of where the program that's being executed is stored. This is called an *absolute file path*. You can use an absolute path if a relative path doesn't work. For instance, if you've put *text_files* in some folder other than *python_work*, then just passing `Path` the path `'text_files/filename.txt'` won't work because Python will only look for that location inside *python_work*. You'll need to write out an absolute path to clarify where you want Python to look.

Absolute paths are usually longer than relative paths, because they start at your system's root folder:

```
path = Path('/home/eric/data_files/text_files/filename.txt')
```

Using absolute paths, you can read files from any location on your system. For now it's easiest to store files in the same directory as your program files, or in a folder such as *text_files* within the directory that stores your program files.

NOTE *Windows systems use a backslash (\) instead of a forward slash (/) when displaying file paths, but you should use forward slashes in your code, even on Windows. The `pathlib` library will automatically use the correct representation of the path when it interacts with your system, or any user's system.*

Accessing a File's Lines

When you're working with a file, you'll often want to examine each line of the file. You might be looking for certain information in the file, or

you might want to modify the text in the file in some way. For example, you might want to read through a file of weather data and work with any line that includes the word *sunny* in the description of that day's weather. In a news report, you might look for any line with the tag `<headline>` and rewrite that line with a specific kind of formatting.

You can use the `splitlines()` method to turn a long string into a set of lines, and then use a `for` loop to examine each line from a file, one at a time:

file_reader.py

```
from pathlib import Path

path = Path('pi_digits.txt')
❶ contents = path.read_text()

❷ lines = contents.splitlines()
for line in lines:
    print(line)
```

We start out by reading the entire contents of the file, as we did earlier ❶. If you're planning to work with the individual lines in a file, you don't need to strip any whitespace when reading the file. The `splitlines()` method returns a list of all lines in the file, and we assign this list to the variable `lines` ❷. We then loop over these lines and print each one:

```
3.1415926535
  8979323846
  2643383279
```

Since we haven't modified any of the lines, the output matches the original text file exactly.

Working with a File's Contents

After you've read the contents of a file into memory, you can do whatever you want with that data, so let's briefly explore the digits of *pi*. First, we'll attempt to build a single string containing all the digits in the file with no whitespace in it:

pi_string.py

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
❶ for line in lines:
    pi_string += line

print(pi_string)
print(len(pi_string))
```

We start by reading the file and storing each line of digits in a list, just as we did in the previous example. We then create a variable, `pi_string`,

to hold the digits of *pi*. We write a loop that adds each line of digits to `pi_string` ❶. We print this string, and also show how long the string is:

```
3.1415926535  8979323846  2643383279
36
```

The variable `pi_string` contains the whitespace that was on the left side of the digits in each line, but we can get rid of that by using `lstrip()` on each line:

```
--snip--
for line in lines:
    pi_string += line.lstrip()

print(pi_string)
print(len(pi_string))
```

Now we have a string containing *pi* to 30 decimal places. The string is 32 characters long because it also includes the leading `3` and a decimal point:

```
3.14159265358979323846264338327 9
32
```

NOTE *When Python reads from a text file, it interprets all text in the file as a string. If you read in a number and want to work with that value in a numerical context, you'll have to convert it to an integer using the `int()` function or a float using the `float()` function.*

Large Files: One Million Digits

So far, we've focused on analyzing a text file that contains only three lines, but the code in these examples would work just as well on much larger files. If we start with a text file that contains *pi* to 1,000,000 decimal places, instead of just 30, we can create a single string containing all these digits. We don't need to change our program at all, except to pass it a different file. We'll also print just the first 50 decimal places, so we don't have to watch a million digits scroll by in the terminal:

pi_string.py

```
from pathlib import Path

path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

The output shows that we do indeed have a string containing *pi* to 1,000,000 decimal places:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

Python has no inherent limit to how much data you can work with; you can work with as much data as your system's memory can handle.

NOTE *To run this program (and many of the examples that follow), you'll need to download the resources available at* https://ehmatthes.github.io/pcc_3e.

Is Your Birthday Contained in Pi?

I've always been curious to know if my birthday appears anywhere in the digits of *pi*. Let's use the program we just wrote to find out if someone's birthday appears anywhere in the first million digits of *pi*. We can do this by expressing each birthday as a string of digits and seeing if that string appears anywhere in `pi_string`:

pi_birthday.py

```
--snip--
for line in lines:
    pi_string += line.strip()

birthday = input("Enter your birthday, in the form mmddyy: ")
if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of pi.")
```

We first prompt for the user's birthday, and then check if that string is in `pi_string`. Let's try it:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

My birthday does appear in the digits of *pi*! Once you've read from a file, you can analyze its contents in just about any way you can imagine.

TRY IT YOURSELF

10-1. Learning Python: Open a blank file in your text editor and write a few lines summarizing what you've learned about Python so far. Start each line with the phrase *In Python you can. . .* . Save the file as *learning_python.txt* in the same directory as your exercises from this chapter. Write a program that reads the file and prints what you wrote two times: print the contents once by reading in the entire file, and once by storing the lines in a list and then looping over each line.

(continued)

10-2. Learning C: You can use the `replace()` method to replace any word in a string with a different word. Here's a quick example showing how to replace `'dog'` with `'cat'` in a sentence:

```
>>> message = "I really like dogs."
>>> message.replace('dog', 'cat')
'I really like cats.'
```

Read in each line from the file you just created, *learning_python.txt*, and replace the word *Python* with the name of another language, such as *C*. Print each modified line to the screen.

10-3. Simpler Code: The program *file_reader.py* in this section uses a temporary variable, `lines`, to show how `splitlines()` works. You can skip the temporary variable and loop directly over the list that `splitlines()` returns:

```
for line in contents.splitlines():
```

Remove the temporary variable from each of the programs in this section, to make them more concise.

Writing to a File

One of the simplest ways to save data is to write it to a file. When you write text to a file, the output will still be available after you close the terminal containing your program's output. You can examine output after a program finishes running, and you can share the output files with others as well. You can also write programs that read the text back into memory and work with it again later.

Writing a Single Line

Once you have a path defined, you can write to a file using the `write_text()` method. To see how this works, let's write a simple message and store it in a file instead of printing it to the screen:

write_message.py

```
from pathlib import Path

path = Path('programming.txt')
path.write_text("I love programming.")
```

The `write_text()` method takes a single argument: the string that you want to write to the file. This program has no terminal output, but if you open the file *programming.txt*, you'll see one line:

programming.txt

```
I love programming.
```

This file behaves like any other file on your computer. You can open it, write new text in it, copy from it, paste to it, and so forth.

NOTE *Python can only write strings to a text file. If you want to store numerical data in a text file, you'll have to convert the data to string format first using the* `str()` *function.*

Writing Multiple Lines

The `write_text()` method does a few things behind the scenes. If the file that `path` points to doesn't exist, it creates that file. Also, after writing the string to the file, it makes sure the file is closed properly. Files that aren't closed properly can lead to missing or corrupted data.

To write more than one line to a file, you need to build a string containing the entire contents of the file, and then call `write_text()` with that string. Let's write several lines to the *programming.txt* file:

```
from pathlib import Path

contents = "I love programming.\n"
contents += "I love creating new games.\n"
contents += "I also love working with data.\n"

path = Path('programming.txt')
path.write_text(contents)
```

We define a variable called `contents` that will hold the entire contents of the file. On the next line, we use the `+=` operator to add to this string. You can do this as many times as you need, to build strings of any length. In this case we include newline characters at the end of each line, to make sure each statement appears on its own line.

If you run this and then open *programming.txt*, you'll see each of these lines in the text file:

```
I love programming.
I love creating new games.
I also love working with data.
```

You can also use spaces, tab characters, and blank lines to format your output, just as you've been doing with terminal-based output. There's no limit to the length of your strings, and this is how many computer-generated documents are created.

NOTE *Be careful when calling* `write_text()` *on a path object. If the file already exists,* `write_text()` *will erase the current contents of the file and write new contents to the file. Later in this chapter, you'll learn to check whether a file exists using* `pathlib`*.*

TRY IT YOURSELF

10-4. Guest: Write a program that prompts the user for their name. When they respond, write their name to a file called *guest.txt*.

10-5. Guest Book: Write a `while` loop that prompts users for their name. Collect all the names that are entered, and then write these names to a file called *guest_book.txt*. Make sure each entry appears on a new line in the file.

Exceptions

Python uses special objects called *exceptions* to manage errors that arise during a program's execution. Whenever an error occurs that makes Python unsure of what to do next, it creates an exception object. If you write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a *traceback*, which includes a report of the exception that was raised.

Exceptions are handled with `try-except` blocks. A *try-except* block asks Python to do something, but it also tells Python what to do if an exception is raised. When you use `try-except` blocks, your programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you've written.

Handling the ZeroDivisionError Exception

Let's look at a simple error that causes Python to raise an exception. You probably know that it's impossible to divide a number by zero, but let's ask Python to do it anyway:

division_calculator.py

```
print(5/0)
```

Python can't do this, so we get a traceback:

```
Traceback (most recent call last):
  File "division_calculator.py", line 1, in <module>
    print(5/0)
          ~^~
❶ ZeroDivisionError: division by zero
```

The error reported in the traceback, `ZeroDivisionError`, is an exception object ❶. Python creates this kind of object in response to a situation where it can't do what we ask it to. When this happens, Python stops the program and tells us the kind of exception that was raised. We can use this information to modify our program. We'll tell Python what to do when this kind of exception occurs; that way, if it happens again, we'll be prepared.

Using try-except Blocks

When you think an error may occur, you can write a `try-except` block to handle the exception that might be raised. You tell Python to try running some code, and you tell it what to do if the code results in a particular kind of exception.

Here's what a `try-except` block for handling the `ZeroDivisionError` exception looks like:

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

We put `print(5/0)`, the line that caused the error, inside a `try` block. If the code in a `try` block works, Python skips over the `except` block. If the code in the `try` block causes an error, Python looks for an `except` block whose error matches the one that was raised, and runs the code in that block.

In this example, the code in the `try` block produces a `ZeroDivisionError`, so Python looks for an `except` block telling it how to respond. Python then runs the code in that block, and the user sees a friendly error message instead of a traceback:

```
You can't divide by zero!
```

If more code followed the `try-except` block, the program would continue running because we told Python how to handle the error. Let's look at an example where catching an error can allow a program to continue running.

Using Exceptions to Prevent Crashes

Handling errors correctly is especially important when the program has more work to do after the error occurs. This happens often in programs that prompt users for input. If the program responds to invalid input appropriately, it can prompt for more valid input instead of crashing.

Let's create a simple calculator that does only division:

division_calculator.py

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
❶   first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
❷   second_number = input("Second number: ")
    if second_number == 'q':
        break
❸   answer = int(first_number) / int(second_number)
    print(answer)
```

This program prompts the user to input a `first_number` ❶ and, if the user does not enter `q` to quit, a `second_number` ❷. We then divide these two numbers to get an `answer` ❸. This program does nothing to handle errors, so asking it to divide by zero causes it to crash:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "division_calculator.py", line 11, in <module>
    answer = int(first_number) / int(second_number)
             ~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~
ZeroDivisionError: division by zero
```

It's bad that the program crashed, but it's also not a good idea to let users see tracebacks. Nontechnical users will be confused by them, and in a malicious setting, attackers will learn more than you want them to. For example, they'll know the name of your program file, and they'll see a part of your code that isn't working properly. A skilled attacker can sometimes use this information to determine which kind of attacks to use against your code.

The else Block

We can make this program more error resistant by wrapping the line that might produce errors in a `try-except` block. The error occurs on the line that performs the division, so that's where we'll put the `try-except` block. This example also includes an `else` block. Any code that depends on the `try` block executing successfully goes in the `else` block:

```
--snip--
while True:
    --snip--
    if second_number == 'q':
        break
❶   try:
        answer = int(first_number) / int(second_number)
❷   except ZeroDivisionError:
        print("You can't divide by 0!")
❸   else:
        print(answer)
```

We ask Python to try to complete the division operation in a `try` block ❶, which includes only the code that might cause an error. Any code that depends on the `try` block succeeding is added to the `else` block. In this case, if the division operation is successful, we use the `else` block to print the result ❸.

The `except` block tells Python how to respond when a `ZeroDivisionError` arises ❷. If the `try` block doesn't succeed because of a division-by-zero error,

we print a friendly message telling the user how to avoid this kind of error. The program continues to run, and the user never sees a traceback:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
You can't divide by 0!

First number: 5
Second number: 2
2.5

First number: q
```

The only code that should go in a `try` block is code that might cause an exception to be raised. Sometimes you'll have additional code that should run only if the `try` block was successful; this code goes in the `else` block. The `except` block tells Python what to do in case a certain exception arises when it tries to run the code in the `try` block.

By anticipating likely sources of errors, you can write robust programs that continue to run even when they encounter invalid data and missing resources. Your code will be resistant to innocent user mistakes and malicious attacks.

Handling the FileNotFoundError Exception

One common issue when working with files is handling missing files. The file you're looking for might be in a different location, the filename might be misspelled, or the file might not exist at all. You can handle all of these situations with a `try-except` block.

Let's try to read a file that doesn't exist. The following program tries to read in the contents of *Alice in Wonderland*, but I haven't saved the file *alice.txt* in the same directory as *alice.py*:

alice.py

```
from pathlib import Path

path = Path('alice.txt')
contents = path.read_text(encoding='utf-8')
```

Note that we're using `read_text()` in a slightly different way here than what you saw earlier. The `encoding` argument is needed when your system's default encoding doesn't match the encoding of the file that's being read. This is most likely to happen when reading from a file that wasn't created on your system.

Python can't read from a missing file, so it raises an exception:

```
Traceback (most recent call last):
❶  File "alice.py", line 4, in <module>
❷    contents = path.read_text(encoding='utf-8')
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
  File "/.../pathlib.py", line 1056, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/.../pathlib.py", line 1042, in open
    return io.open(self, mode, buffering, encoding, errors, newline)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
❸ FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

This is a longer traceback than the ones we've seen previously, so let's look at how you can make sense of more complex tracebacks. It's often best to start at the very end of the traceback. On the last line, we can see that a `FileNotFoundError` exception was raised ❸. This is important because it tells us what kind of exception to use in the `except` block that we'll write.

Looking back near the beginning of the traceback ❶, we can see that the error occurred at line 4 in the file *alice.py*. The next line shows the line of code that caused the error ❷. The rest of the traceback shows some code from the libraries that are involved in opening and reading from files. You don't usually need to read through or understand all of these lines in a traceback.

To handle the error that's being raised, the `try` block will begin with the line that was identified as problematic in the traceback. In our example, this is the line that contains `read_text()`:

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
❶ except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
```

In this example, the code in the `try` block produces a `FileNotFoundError`, so we write an `except` block that matches that error ❶. Python then runs the code in that block when the file can't be found, and the result is a friendly error message instead of a traceback:

```
Sorry, the file alice.txt does not exist.
```

The program has nothing more to do if the file doesn't exist, so this is all the output we see. Let's build on this example and see how exception handling can help when you're working with more than one file.

Analyzing Text

You can analyze text files containing entire books. Many classic works of literature are available as simple text files because they are in the public domain. The texts used in this section come from Project Gutenberg (*https://gutenberg.org*). Project Gutenberg maintains a collection of literary works that are available in the public domain, and it's a great resource if you're interested in working with literary texts in your programming projects.

Let's pull in the text of *Alice in Wonderland* and try to count the number of words in the text. To do this, we'll use the string method `split()`, which by default splits a string wherever it finds any whitespace:

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
else:
    # Count the approximate number of words in the file:
❶   words = contents.split()
❷   num_words = len(words)
    print(f"The file {path} has about {num_words} words.")
```

I moved the file *alice.txt* to the correct directory, so the `try` block will work this time. We take the string `contents`, which now contains the entire text of *Alice in Wonderland* as one long string, and use `split()` to produce a list of all the words in the book ❶. Using `len()` on this list ❷ gives us a good approximation of the number of words in the original text. Lastly, we print a statement that reports how many words were found in the file. This code is placed in the `else` block because it only works if the code in the `try` block was executed successfully.

The output tells us how many words are in *alice.txt*:

```
The file alice.txt has about 29594 words.
```

The count is a little high because extra information is provided by the publisher in the text file used here, but it's a good approximation of the length of *Alice in Wonderland*.

Working with Multiple Files

Let's add more books to analyze, but before we do, let's move the bulk of this program to a function called `count_words()`. This will make it easier to run the analysis for multiple books:

word_count.py

```
from pathlib import Path

def count_words(path):
❶   """Count the approximate number of words in a file."""
    try:
        contents = path.read_text(encoding='utf-8')
    except FileNotFoundError:
        print(f"Sorry, the file {path} does not exist.")
    else:
        # Count the approximate number of words in the file:
        words = contents.split()
        num_words = len(words)
        print(f"The file {path} has about {num_words} words.")
```

```
path = Path('alice.txt')
count_words(path)
```

Most of this code is unchanged. It's only been indented, and moved into the body of `count_words()`. It's a good habit to keep comments up to date when you're modifying a program, so the comment has also been changed to a docstring and reworded slightly ❶.

Now we can write a short loop to count the words in any text we want to analyze. We do this by storing the names of the files we want to analyze in a list, and then we call `count_words()` for each file in the list. We'll try to count the words for *Alice in Wonderland*, *Siddhartha*, *Moby Dick*, and *Little Women*, which are all available in the public domain. I've intentionally left *siddhartha.txt* out of the directory containing *word_count.py*, so we can see how well our program handles a missing file:

```
from pathlib import Path

def count_words(filename):
    --snip--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
        'little_women.txt']
for filename in filenames:
❶     path = Path(filename)
    count_words(path)
```

The names of the files are stored as simple strings. Each string is then converted to a `Path` object ❶, before the call to `count_words()`. The missing *siddhartha.txt* file has no effect on the rest of the program's execution:

```
The file alice.txt has about 29594 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215864 words.
The file little_women.txt has about 189142 words.
```

Using the `try-except` block in this example provides two significant advantages. We prevent our users from seeing a traceback, and we let the program continue analyzing the texts it's able to find. If we don't catch the `FileNotFoundError` that *siddhartha.txt* raises, the user would see a full traceback, and the program would stop running after trying to analyze *Siddhartha*. It would never analyze *Moby Dick* or *Little Women*.

Failing Silently

In the previous example, we informed our users that one of the files was unavailable. But you don't need to report every exception you catch. Sometimes, you'll want the program to fail silently when an exception occurs and continue on as if nothing happened. To make a program fail silently, you write a `try` block as usual, but you explicitly tell Python to do

nothing in the except block. Python has a pass statement that tells it to do nothing in a block:

```
def count_words(path):
    """Count the approximate number of words in a file."""
    try:
        --snip--
    except FileNotFoundError:
        pass
    else:
        --snip--
```

The only difference between this listing and the previous one is the pass statement in the except block. Now when a FileNotFoundError is raised, the code in the except block runs, but nothing happens. No traceback is produced, and there's no output in response to the error that was raised. Users see the word counts for each file that exists, but they don't see any indication that a file wasn't found:

```
The file alice.txt has about 29594 words.
The file moby_dick.txt has about 215864 words.
The file little_women.txt has about 189142 words.
```

The pass statement also acts as a placeholder. It's a reminder that you're choosing to do nothing at a specific point in your program's execution and that you might want to do something there later. For example, in this program we might decide to write any missing filenames to a file called *missing_files.txt*. Our users wouldn't see this file, but we'd be able to read the file and deal with any missing texts.

Deciding Which Errors to Report

How do you know when to report an error to your users and when to let your program fail silently? If users know which texts are supposed to be analyzed, they might appreciate a message informing them why some texts were not analyzed. If users expect to see some results but don't know which books are supposed to be analyzed, they might not need to know that some texts were unavailable. Giving users information they aren't looking for can decrease the usability of your program. Python's error-handling structures give you fine-grained control over how much to share with users when things go wrong; it's up to you to decide how much information to share.

Well-written, properly tested code is not very prone to internal errors, such as syntax or logical errors. But every time your program depends on something external such as user input, the existence of a file, or the availability of a network connection, there is a possibility of an exception being raised. A little experience will help you know where to include exception-handling blocks in your program and how much to report to users about errors that arise.

TRY IT YOURSELF

10-6. Addition: One common problem when prompting for numerical input occurs when people provide text instead of numbers. When you try to convert the input to an `int`, you'll get a `ValueError`. Write a program that prompts for two numbers. Add them together and print the result. Catch the `ValueError` if either input value is not a number, and print a friendly error message. Test your program by entering two numbers and then by entering some text instead of a number.

10-7. Addition Calculator: Wrap your code from Exercise 10-5 in a `while` loop so the user can continue entering numbers, even if they make a mistake and enter text instead of a number.

10-8. Cats and Dogs: Make two files, *cats.txt* and *dogs.txt*. Store at least three names of cats in the first file and three names of dogs in the second file. Write a program that tries to read these files and print the contents of the file to the screen. Wrap your code in a `try-except` block to catch the `FileNotFound` error, and print a friendly message if a file is missing. Move one of the files to a different location on your system, and make sure the code in the `except` block executes properly.

10-9. Silent Cats and Dogs: Modify your `except` block in Exercise 10-7 to fail silently if either file is missing.

10-10. Common Words: Visit Project Gutenberg (*https://gutenberg.org*) and find a few texts you'd like to analyze. Download the text files for these works, or copy the raw text from your browser into a text file on your computer.

You can use the `count()` method to find out how many times a word or phrase appears in a string. For example, the following code counts the number of times `'row'` appears in a string:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Notice that converting the string to lowercase using `lower()` catches all appearances of the word you're looking for, regardless of how it's formatted.

Write a program that reads the files you found at Project Gutenberg and determines how many times the word `'the'` appears in each text. This will be an approximation because it will also count words such as `'then'` and `'there'`. Try counting `'the '`, with a space in the string, and see how much lower your count is.

Storing Data

Many of your programs will ask users to input certain kinds of information. You might allow users to store preferences in a game or provide data for a visualization. Whatever the focus of your program is, you'll store the information users provide in data structures such as lists and dictionaries. When users close a program, you'll almost always want to save the information they entered. A simple way to do this involves storing your data using the `json` module.

The `json` module allows you to convert simple Python data structures into JSON-formatted strings, and then load the data from that file the next time the program runs. You can also use `json` to share data between different Python programs. Even better, the JSON data format is not specific to Python, so you can share data you store in the JSON format with people who work in many other programming languages. It's a useful and portable format, and it's easy to learn.

NOTE *The* JSON (JavaScript Object Notation) *format was originally developed for JavaScript. However, it has since become a common format used by many languages, including Python.*

Using json.dumps() and json.loads()

Let's write a short program that stores a set of numbers and another program that reads these numbers back into memory. The first program will use `json.dumps()` to store the set of numbers, and the second program will use `json.loads()`.

The `json.dumps()` function takes one argument: a piece of data that should be converted to the JSON format. The function returns a string, which we can then write to a data file:

number_writer.py

```
from pathlib import Path
import json

numbers = [2, 3, 5, 7, 11, 13]

❶ path = Path('numbers.json')
❷ contents = json.dumps(numbers)
path.write_text(contents)
```

We first import the `json` module, and then create a list of numbers to work with. Then we choose a filename in which to store the list of numbers ❶. It's customary to use the file extension *.json* to indicate that the data in the file is stored in the JSON format. Next, we use the `json.dumps()` ❷ function to generate a string containing the JSON representation of the data we're working with. Once we have this string, we write it to the file using the same `write_text()` method we used earlier.

This program has no output, but let's open the file *numbers.json* and look at it. The data is stored in a format that looks just like Python:

```
[2, 3, 5, 7, 11, 13]
```

Now we'll write a separate program that uses `json.loads()` to read the list back into memory:

number_reader.py

```
from pathlib import Path
import json

❶ path = Path('numbers.json')
❷ contents = path.read_text()
❸ numbers = json.loads(contents)

print(numbers)
```

We make sure to read from the same file we wrote to ❶. Since the data file is just a text file with specific formatting, we can read it with the `read_text()` method ❷. We then pass the contents of the file to `json.loads()` ❸. This function takes in a JSON-formatted string and returns a Python object (in this case, a list), which we assign to `numbers`. Finally, we print the recovered list of numbers and see that it's the same list created in *number_writer.py*:

```
[2, 3, 5, 7, 11, 13]
```

This is a simple way to share data between two programs.

Saving and Reading User-Generated Data

Saving data with `json` is useful when you're working with user-generated data, because if you don't store your user's information somehow, you'll lose it when the program stops running. Let's look at an example where we prompt the user for their name the first time they run a program and then remember their name when they run the program again.

Let's start by storing the user's name:

remember_me.py

```
from pathlib import Path
import json

❶ username = input("What is your name? ")

❷ path = Path('username.json')
  contents = json.dumps(username)
  path.write_text(contents)

❸ print(f"We'll remember you when you come back, {username}!")
```

We first prompt for a username to store ❶. Next, we write the data we just collected to a file called *username.json* ❷. Then we print a message informing the user that we've stored their information ❸:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Now let's write a new program that greets a user whose name has already been stored:

greet_user.py

```
from pathlib import Path
import json

❶ path = Path('username.json')
  contents = path.read_text()
❷ username = json.loads(contents)

  print(f"Welcome back, {username}!")
```

We read the contents of the data file ❶ and then use `json.loads()` to assign the recovered data to the variable `username` ❷. Since we've recovered the username, we can welcome the user back with a personalized greeting:

```
Welcome back, Eric!
```

We need to combine these two programs into one file. When someone runs *remember_me.py*, we want to retrieve their username from memory if possible; if not, we'll prompt for a username and store it in *username.json* for next time. We could write a `try-except` block here to respond appropriately if *username.json* doesn't exist, but instead we'll use a handy method from the `pathlib` module:

remember_me.py

```
from pathlib import Path
import json

path = Path('username.json')
❶ if path.exists():
    contents = path.read_text()
    username = json.loads(contents)
    print(f"Welcome back, {username}!")
❷ else:
    username = input("What is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    print(f"We'll remember you when you come back, {username}!")
```

There are many helpful methods you can use with `Path` objects. The `exists()` method returns `True` if a file or folder exists and `False` if it doesn't. Here we use `path.exists()` to find out if a username has already been stored ❶. If *username.json* exists, we load the username and print a personalized greeting to the user.

If the file *username.json* doesn't exist ❷, we prompt for a username and store the value that the user enters. We also print the familiar message that we'll remember them when they come back.

Whichever block executes, the result is a username and an appropriate greeting. If this is the first time the program runs, this is the output:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Otherwise:

```
Welcome back, Eric!
```

This is the output you see if the program was already run at least once. Even though the data in this section is just a single string, the program would work just as well with any data that can be converted to a JSON-formatted string.

Refactoring

Often, you'll come to a point where your code will work, but you'll recognize that you could improve the code by breaking it up into a series of functions that have specific jobs. This process is called *refactoring*. Refactoring makes your code cleaner, easier to understand, and easier to extend.

We can refactor *remember_me.py* by moving the bulk of its logic into one or more functions. The focus of *remember_me.py* is on greeting the user, so let's move all of our existing code into a function called `greet_user()`:

remember_me.py

```
from pathlib import Path
import json

def greet_user():
    """Greet the user by name."""
    path = Path('username.json')
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Because we're using a function now, we rewrite the comments as a docstring that reflects how the program currently works ❶. This file is a little cleaner, but the function `greet_user()` is doing more than just greeting the user—it's also retrieving a stored username if one exists and prompting for a new username if one doesn't.

Let's refactor `greet_user()` so it's not doing so many different tasks. We'll start by moving the code for retrieving a stored username to a separate function:

```
from pathlib import Path
import json

def get_stored_username(path):
    """Get stored username if available."""
```

```
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        return username
    else:
❷       return None

def greet_user():
    """Greet the user by name."""
    path = Path('username.json')
    username = get_stored_username(path)
❸   if username:
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

The new function `get_stored_username()` ❶ has a clear purpose, as stated in the docstring. This function retrieves a stored username and returns the username if it finds one. If the path that's passed to `get_stored_username()` doesn't exist, the function returns `None` ❷. This is good practice: a function should either return the value you're expecting, or it should return `None`. This allows us to perform a simple test with the return value of the function. We print a welcome back message to the user if the attempt to retrieve a username is successful ❸, and if it isn't, we prompt for a new username.

We should factor one more block of code out of `greet_user()`. If the username doesn't exist, we should move the code that prompts for a new username to a function dedicated to that purpose:

```
from pathlib import Path
import json

def get_stored_username(path):
    """Get stored username if available."""
    --snip--

def get_new_username(path):
    """Prompt for a new username."""
    username = input("What is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    return username

def greet_user():
    """Greet the user by name."""
    path = Path('username.json')
❶   username = get_stored_username(path)
    if username:
        print(f"Welcome back, {username}!")
```

```
    else:
❷       username = get_new_username(path)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Each function in this final version of *remember_me.py* has a single, clear purpose. We call `greet_user()`, and that function prints an appropriate message: it either welcomes back an existing user or greets a new user. It does this by calling `get_stored_username()` ❶, which is responsible only for retrieving a stored username if one exists. Finally, if necessary, `greet_user()` calls `get_new_username()` ❷, which is responsible only for getting a new username and storing it. This compartmentalization of work is an essential part of writing clear code that will be easy to maintain and extend.

TRY IT YOURSELF

10-11. Favorite Number: Write a program that prompts for the user's favorite number. Use `json.dumps()` to store this number in a file. Write a separate program that reads in this value and prints the message "I know your favorite number! It's _____."

10-12. Favorite Number Remembered: Combine the two programs you wrote in Exercise 10-11 into one file. If the number is already stored, report the favorite number to the user. If not, prompt for the user's favorite number and store it in a file. Run the program twice to see that it works.

10-13. User Dictionary: The *remember_me.py* example only stores one piece of information, the username. Expand this example by asking for two more pieces of information about the user, then store all the information you collect in a dictionary. Write this dictionary to a file using `json.dumps()`, and read it back in using `json.loads()`. Print a summary showing exactly what your program remembers about the user.

10-14. Verify User: The final listing for *remember_me.py* assumes either that the user has already entered their username or that the program is running for the first time. We should modify it in case the current user is not the person who last used the program.

Before printing a welcome back message in `greet_user()`, ask the user if this is the correct username. If it's not, call `get_new_username()` to get the correct username.

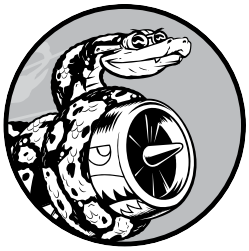
Summary

In this chapter, you learned how to work with files. You learned to read the entire contents of a file, and then work through the contents one line at a time if you need to. You learned to write as much text as you want to a file. You also read about exceptions and how to handle the exceptions you're likely to see in your programs. Finally, you learned how to store Python data structures so you can save information your users provide, preventing them from having to start over each time they run a program.

In Chapter 11, you'll learn efficient ways to test your code. This will help you trust that the code you develop is correct, and it will help you identify bugs that are introduced as you continue to build on the programs you've written.

11

TESTING YOUR CODE



When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in response to all the kinds of input it's designed to receive. When you write tests, you can be confident that your code will work correctly as more people begin to use your programs. You'll also be able to test new code as you add it, to make sure your changes don't break your program's existing behavior. Every programmer makes mistakes, so every programmer must test their code often, to catch problems before users encounter them.

In this chapter, you'll learn to test your code using `pytest`. The `pytest` library is a collection of tools that will help you write your first tests quickly and simply, while supporting your tests as they grow in complexity along with your projects. Python doesn't include `pytest` by default, so you'll learn to install external libraries. Knowing how to install external libraries will make a wide variety of well-designed code available to you. These libraries will expand the kinds of projects you can work on immensely.

You'll learn to build a series of tests and check that each set of inputs results in the output you want. You'll see what a passing test looks like and what a failing test looks like, and you'll learn how a failing test can help you improve your code. You'll learn to test functions and classes, and you'll start to understand how many tests to write for a project.

Installing pytest with pip

While Python includes a lot of functionality in the standard library, Python developers also depend heavily on third-party packages. A *third-party package* is a library that's developed outside the core Python language. Some popular third-party libraries are eventually adopted into the standard library, and end up being included in most Python installations from that point forward. This happens most often with libraries that are unlikely to change much once they've had their initial bugs worked out. These kinds of libraries can evolve at the same pace as the overall language.

Many packages, however, are kept out of the standard library so they can be developed on a timeline independent of the language itself. These packages tend to be updated more frequently than they would be if they were tied to Python's development schedule. This is true of `pytest` and most of the libraries we'll use in the second half of this book. You shouldn't blindly trust every third-party package, but you also shouldn't be put off by the fact that a lot of important functionality is implemented through such packages.

Updating pip

Python includes a tool called pip that's used to install third-party packages. Because pip helps install packages from external resources, it's updated often to address potential security issues. So, we'll start by updating pip.

Open a new terminal window and issue the following command:

```
$ python -m pip install --upgrade pip
❶ Requirement already satisfied: pip in /.../python3.11/site-packages (22.0.4)
--snip--
❷ Successfully installed pip-22.1.2
```

The first part of this command, **`python -m pip`**, tells Python to run the module `pip`. The second part, **`install --upgrade`**, tells pip to update a package that's already been installed. The last part, **`pip`**, specifies which third-party package should be updated. The output shows that my current version of pip, version 22.0.4 ❶, was replaced by the latest version at the time of this writing, 22.1.2 ❷.

You can use this command to update any third-party package installed on your system:

```
$ python -m pip install --upgrade package_name
```

NOTE *If you're using Linux, pip may not be included with your installation of Python. If you get an error when trying to upgrade pip, see the instructions in Appendix A.*

Installing pytest

Now that pip is up to date, we can install `pytest`:

```
$ python -m pip install --user pytest
Collecting pytest
  --snip--
Successfully installed attrs-21.4.0 iniconfig-1.1.1 ...pytest-7.x.x
```

We're still using the core command **`pip install`**, without the **`--upgrade`** flag this time. Instead, we're using the **`--user`** flag, which tells Python to install this package for the current user only. The output shows that the latest version of `pytest` was successfully installed, along with a number of other packages that `pytest` depends on.

You can use this command to install many third-party packages:

```
$ python -m pip install --user package_name
```

NOTE *If you have any difficulty running this command, try running the same command without the* ***`--user`*** *flag.*

Testing a Function

To learn about testing, we need code to test. Here's a simple function that takes in a first and last name, and returns a neatly formatted full name:

name_function.py

```
def get_formatted_name(first, last):
    """Generate a neatly formatted full name."""
    full_name = f"{first} {last}"
    return full_name.title()
```

The function `get_formatted_name()` combines the first and last name with a space in between to complete a full name, and then capitalizes and returns the full name. To check that `get_formatted_name()` works, let's make a program that uses this function. The program *names.py* lets users enter a first and last name, and see a neatly formatted full name:

names.py

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

This program imports `get_formatted_name()` from *name_function.py*. The user can enter a series of first and last names and see the formatted full names that are generated:

```
Enter 'q' at any time to quit.

Please give me a first name: janis
Please give me a last name: joplin
       Neatly formatted name: Janis Joplin.

Please give me a first name: bob
Please give me a last name: dylan
        Neatly formatted name: Bob Dylan.

Please give me a first name: q
```

We can see that the names generated here are correct. But say we want to modify `get_formatted_name()` so it can also handle middle names. As we do so, we want to make sure we don't break the way the function handles names that have only a first and last name. We could test our code by running *names.py* and entering a name like `Janis Joplin` every time we modify `get_formatted_name()`, but that would become tedious. Fortunately, `pytest` provides an efficient way to automate the testing of a function's output. If we automate the testing of `get_formatted_name()`, we can always be confident that the function will work when given the kinds of names we've written tests for.

Unit Tests and Test Cases

There is a wide variety of approaches to testing software. One of the simplest kinds of test is a unit test. A *unit test* verifies that one specific aspect of a function's behavior is correct. A *test case* is a collection of unit tests that together prove that a function behaves as it's supposed to, within the full range of situations you expect it to handle.

A good test case considers all the possible kinds of input a function could receive and includes tests to represent each of these situations. A test case with *full coverage* includes a full range of unit tests covering all the possible ways you can use a function. Achieving full coverage on a large project can be daunting. It's often good enough to write tests for your code's critical behaviors and then aim for full coverage only if the project starts to see widespread use.

A Passing Test

With `pytest`, writing your first unit test is pretty straightforward. We'll write a single test function. The test function will call the function we're testing, and we'll make an assertion about the value that's returned. If our assertion is correct, the test will pass; if the assertion is incorrect, the test will fail.

Here's the first test of the function `get_formatted_name()`:

test_name_function.py

```
from name_function import get_formatted_name

❶ def test_first_last_name():
      """Do names like 'Janis Joplin' work?"""
❷     formatted_name = get_formatted_name('janis', 'joplin')
❸     assert formatted_name == 'Janis Joplin'
```

Before we run the test, let's take a closer look at this function. The name of a test file is important; it must start with *test_*. When we ask `pytest` to run the tests we've written, it will look for any file that begins with *test_*, and run all of the tests it finds in that file.

In the test file, we first import the function that we want to test: `get_formatted_name()`. Then we define a test function: in this case, `test_first_last_name()` ❶. This is a longer function name than we've been using, for a good reason. First, test functions need to start with the word *test*, followed by an underscore. Any function that starts with `test_` will be *discovered* by `pytest`, and will be run as part of the testing process.

Also, test names should be longer and more descriptive than a typical function name. You'll never call the function yourself; `pytest` will find the function and run it for you. Test function names should be long enough that if you see the function name in a test report, you'll have a good sense of what behavior was being tested.

Next, we call the function we're testing ❷. Here we call `get_formatted_name()` with the arguments `'janis'` and `'joplin'`, just like we used when we ran *names.py*. We assign the return value of this function to `formatted_name`.

Finally, we make an assertion ❸. An *assertion* is a claim about a condition. Here we're claiming that the value of `formatted_name` should be `'Janis Joplin'`.

Running a Test

If you run the file *test_name_function.py* directly, you won't get any output because we never called the test function. Instead, we'll have `pytest` run the test file for us.

To do this, open a terminal window and navigate to the folder that contains the test file. If you're using VS Code, you can open the folder containing the test file and use the terminal that's embedded in the editor window. In the terminal window, enter the command **`pytest`**. Here's what you should see:

```
$ pytest
======================== test session starts ========================
❶ platform darwin -- Python 3.x.x, pytest-7.x.x, pluggy-1.x.x
❷ rootdir: /.../python_work/chapter_11
❸ collected 1 item

❹ test_name_function.py .                                      [100%]
========================= 1 passed in 0.00s =========================
```

Let's try to make sense of this output. First of all, we see some information about the system the test is running on ❶. I'm testing this on a macOS system, so you may see some different output here. Most importantly, we can see which versions of Python, `pytest`, and other packages are being used to run the test.

Next, we see the directory where the test is being run from ❷: in my case, *python_work/chapter_11*. We can see that `pytest` found one test to run ❸, and we can see the test file that's being run ❹. The single dot after the name of the file tells us that a single test passed, and the `100%` makes it clear that all of the tests have been run. A large project can have hundreds or thousands of tests, and the dots and percentage-complete indicator can be helpful in monitoring the overall progress of the test run.

The last line tells us that one test passed, and it took less than 0.01 seconds to run the test.

This output indicates that the function `get_formatted_name()` will always work for names that have a first and last name, unless we modify the function. When we modify `get_formatted_name()`, we can run this test again. If the test passes, we know the function will still work for names like Janis Joplin.

NOTE *If you're not sure how to navigate to the right location in the terminal, see "Running Python Programs from a Terminal" on page 11. Also, if you see a message that the* `pytest` *command was not found, use the command* `python -m pytest` *instead.*

A Failing Test

What does a failing test look like? Let's modify `get_formatted_name()` so it can handle middle names, but let's do so in a way that breaks the function for names with just a first and last name, like Janis Joplin.

Here's a new version of `get_formatted_name()` that requires a middle name argument:

name_function.py

```
def get_formatted_name(first, middle, last):
    """Generate a neatly formatted full name."""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

This version should work for people with middle names, but when we test it, we see that we've broken the function for people with just a first and last name.

This time, running **`pytest`** gives the following output:

```
$ pytest
======================== test session starts ========================
--snip--
❶ test_name_function.py F                                      [100%]
❷ ============================= FAILURES ==============================
❸ _____________________ test_first_last_name ______________________
    def test_first_last_name():
        """Do names like 'Janis Joplin' work?"""
❹ >       formatted_name = get_formatted_name('janis', 'joplin')
❺ E       TypeError: get_formatted_name() missing 1 required positional
              argument: 'last'
```

```
test_name_function.py:5: TypeError
====================== short test summary info ======================
FAILED test_name_function.py::test_first_last_name - TypeError:
    get_formatted_name() missing 1 required positional argument: 'last'
======================== 1 failed in 0.04s ========================
```

There's a lot of information here because there's a lot you might need to know when a test fails. The first item of note in the output is a single `F` ❶, which tells us that one test failed. We then see a section that focuses on `FAILURES` ❷, because failed tests are usually the most important thing to focus on in a test run. Next, we see that `test_first_last_name()` was the test function that failed ❸. An angle bracket ❹ indicates the line of code that caused the test to fail. The `E` on the next line ❺ shows the actual error that caused the failure: a `TypeError` due to a missing required positional argument, `last`. The most important information is repeated in a shorter summary at the end, so when you're running many tests, you can get a quick sense of which tests failed and why.

Responding to a Failed Test

What do you do when a test fails? Assuming you're checking the right conditions, a passing test means the function is behaving correctly and a failing test means there's an error in the new code you wrote. So when a test fails, don't change the test. If you do, your tests might pass, but any code that calls your function like the test does will suddenly stop working. Instead, fix the code that's causing the test to fail. Examine the changes you just made to the function, and figure out how those changes broke the desired behavior.

In this case, `get_formatted_name()` used to require only two parameters: a first name and a last name. Now it requires a first name, middle name, and last name. The addition of that mandatory middle name parameter broke the original behavior of `get_formatted_name()`. The best option here is to make the middle name optional. Once we do, our test for names like `Janis Joplin` should pass again, and we should be able to accept middle names as well. Let's modify `get_formatted_name()` so middle names are optional and then run the test case again. If it passes, we'll move on to making sure the function handles middle names properly.

To make middle names optional, we move the parameter `middle` to the end of the parameter list in the function definition and give it an empty default value. We also add an `if` test that builds the full name properly, depending on whether a middle name is provided:

name_function.py

```
def get_formatted_name(first, last, middle=''):
    """Generate a neatly formatted full name."""
    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"
    return full_name.title()
```

In this new version of `get_formatted_name()`, the middle name is optional. If a middle name is passed to the function, the full name will contain a first, middle, and last name. Otherwise, the full name will consist of just a first and last name. Now the function should work for both kinds of names. To find out if the function still works for names like `Janis Joplin`, let's run the test again:

```
$ pytest
======================== test session starts ========================
--snip--
test_name_function.py .                                      [100%]
========================= 1 passed in 0.00s =========================
```

The test passes now. This is ideal; it means the function works for names like `Janis Joplin` again, without us having to test the function manually. Fixing our function was easier because the failed test helped us identify how the new code broke existing behavior.

Adding New Tests

Now that we know `get_formatted_name()` works for simple names again, let's write a second test for people who include a middle name. We do this by adding another test function to the file *test_name_function.py*:

test_name_function.py

```
from name_function import get_formatted_name

def test_first_last_name():
    --snip--

def test_first_last_middle_name():
    """Do names like 'Wolfgang Amadeus Mozart' work?"""
❶   formatted_name = get_formatted_name(
        'wolfgang', 'mozart', 'amadeus')
❷   assert formatted_name == 'Wolfgang Amadeus Mozart'
```

We name this new function `test_first_last_middle_name()`. The function name must start with `test_` so the function runs automatically when we run **`pytest`**. We name the function to make it clear which behavior of `get_formatted_name()` we're testing. As a result, if the test fails, we'll know right away what kinds of names are affected.

To test the function, we call `get_formatted_name()` with a first, last, and middle name ❶, and then we make an assertion ❷ that the returned full name matches the full name (first, middle, and last) that we expect. When we run **`pytest`** again, both tests pass:

```
$ pytest
======================== test session starts ========================
--snip--
collected 2 items

❶ test_name_function.py ..                                   [100%]
========================= 2 passed in 0.01s =========================
```

The two dots ❶ indicate that two tests passed, which is also clear from the last line of output. This is great! We now know that the function still works for names like `Janis Joplin`, and we can be confident that it will work for names like `Wolfgang Amadeus Mozart` as well.

TRY IT YOURSELF

11-1. City, Country: Write a function that accepts two parameters: a city name and a country name. The function should return a single string of the form *City, Country*, such as `Santiago, Chile`. Store the function in a module called *city_functions.py*, and save this file in a new folder so `pytest` won't try to run the tests we've already written.

Create a file called *test_cities.py* that tests the function you just wrote. Write a function called `test_city_country()` to verify that calling your function with values such as `'santiago'` and `'chile'` results in the correct string. Run the test, and make sure `test_city_country()` passes.

11-2. Population: Modify your function so it requires a third parameter, `population`. It should now return a single string of the form *City, Country – population xxx*, such as `Santiago, Chile – population 5000000`. Run the test again, and make sure `test_city_country()` fails this time.

Modify the function so the `population` parameter is optional. Run the test, and make sure `test_city_country()` passes again.

Write a second test called `test_city_country_population()` that verifies you can call your function with the values `'santiago'`, `'chile'`, and `'population =5000000'`. Run the tests one more time, and make sure this new test passes.

Testing a Class

In the first part of this chapter, you wrote tests for a single function. Now you'll write tests for a class. You'll use classes in many of your own programs, so it's helpful to be able to prove that your classes work correctly. If you have passing tests for a class you're working on, you can be confident that improvements you make to the class won't accidentally break its current behavior.

A Variety of Assertions

So far, you've seen just one kind of assertion: a claim that a string has a specific value. When writing a test, you can make any claim that can be expressed as a conditional statement. If the condition is `True` as expected, your assumption about how that part of your program behaves will be confirmed; you can be confident that no errors exist. If the condition you

assume is `True` is actually `False`, the test will fail and you'll know there's an issue to resolve. Table 11-1 shows some of the most useful kinds of assertions you can include in your initial tests.

Table 11-1: Commonly Used Assertion Statements in Tests

Assertion	Claim
`assert a == b`	Assert that two values are equal.
`assert a != b`	Assert that two values are not equal.
`assert a`	Assert that a evaluates to `True`.
`assert not a`	Assert that a evaluates to `False`.
`assert element in list`	Assert that an element is in a list.
`assert element not in list`	Assert that an element is not in a list.

These are just a few examples; anything that can be expressed as a conditional statement can be included in a test.

A Class to Test

Testing a class is similar to testing a function, because much of the work involves testing the behavior of the methods in the class. However, there are a few differences, so let's write a class to test. Consider a class that helps administer anonymous surveys:

survey.py

```
class AnonymousSurvey:
    """Collect anonymous answers to a survey question."""

❶   def __init__(self, question):
        """Store a question, and prepare to store responses."""
        self.question = question
        self.responses = []

❷   def show_question(self):
        """Show the survey question."""
        print(self.question)

❸   def store_response(self, new_response):
        """Store a single response to the survey."""
        self.responses.append(new_response)

❹   def show_results(self):
        """Show all the responses that have been given."""
        print("Survey results:")
        for response in self.responses:
            print(f"- {response}")
```

This class starts with a survey question that you provide ❶ and includes an empty list to store responses. The class has methods to print the survey question ❷, add a new response to the response list ❸, and print all the responses stored in the list ❹. To create an instance from this class, all you

have to provide is a question. Once you have an instance representing a particular survey, you display the survey question with `show_question()`, store a response using `store_response()`, and show results with `show_results()`.

To show that the `AnonymousSurvey` class works, let's write a program that uses the class:

language_survey.py

```
from survey import AnonymousSurvey

# Define a question, and make a survey.
question = "What language did you first learn to speak?"
language_survey = AnonymousSurvey(question)

# Show the question, and store responses to the question.
language_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    language_survey.store_response(response)

# Show the survey results.
print("\nThank you to everyone who participated in the survey!")
language_survey.show_results()
```

This program defines a question (`"What language did you first learn to speak?"`) and creates an `AnonymousSurvey` object with that question. The program calls `show_question()` to display the question and then prompts for responses. Each response is stored as it is received. When all responses have been entered (the user inputs `q` to quit), `show_results()` prints the survey results:

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- English
- Spanish
- English
- Mandarin
```

This class works for a simple anonymous survey, but say we want to improve `AnonymousSurvey` and the module it's in, `survey`. We could allow each user to enter more than one response, we could write a method to list only unique responses and to report how many times each response was given, or we could even write another class to manage non-anonymous surveys.

Implementing such changes would risk affecting the current behavior of the class `AnonymousSurvey`. For example, it's possible that while trying to allow each user to enter multiple responses, we could accidentally change how single responses are handled. To ensure we don't break existing behavior as we develop this module, we can write tests for the class.

Testing the AnonymousSurvey Class

Let's write a test that verifies one aspect of the way `AnonymousSurvey` behaves. We'll write a test to verify that a single response to the survey question is stored properly:

test_survey.py

```
from survey import AnonymousSurvey

❶ def test_store_single_response():
      """Test that a single response is stored properly."""
      question = "What language did you first learn to speak?"
❷     language_survey = AnonymousSurvey(question)
      language_survey.store_response('English')
❸     assert 'English' in language_survey.responses
```

We start by importing the class we want to test, `AnonymousSurvey`. The first test function will verify that when we store a response to the survey question, the response will end up in the survey's list of responses. A good descriptive name for this function is `test_store_single_response()` ❶. If this test fails, we'll know from the function name in the test summary that there was a problem storing a single response to the survey.

To test the behavior of a class, we need to make an instance of the class. We create an instance called `language_survey` ❷ with the question `"What language did you first learn to speak?"` We store a single response, `English`, using the `store_response()` method. Then we verify that the response was stored correctly by asserting that `English` is in the list `language_survey.responses` ❸.

By default, running the command **`pytest`** with no arguments will run all the tests that `pytest` discovers in the current directory. To focus on the tests in one file, pass the name of the test file you want to run. Here we'll run just the one test we wrote for `AnonymousSurvey`:

```
$ pytest test_survey.py
======================== test session starts ========================
--snip--
test_survey.py .                                                [100%]
========================= 1 passed in 0.01s =========================
```

This is a good start, but a survey is useful only if it generates more than one response. Let's verify that three responses can be stored correctly. To do this, we add another method to `TestAnonymousSurvey`:

```
from survey import AnonymousSurvey

def test_store_single_response():
```

```
    --snip--

def test_store_three_responses():
    """Test that three individual responses are stored properly."""
    question = "What language did you first learn to speak?"
    language_survey = AnonymousSurvey(question)
❶   responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
        language_survey.store_response(response)

❷   for response in responses:
        assert response in language_survey.responses
```

We call the new function `test_store_three_responses()`. We create a survey object just like we did in `test_store_single_response()`. We define a list containing three different responses ❶, and then we call `store_response()` for each of these responses. Once the responses have been stored, we write another loop and assert that each response is now in `language_survey.responses` ❷.

When we run the test file again, both tests (for a single response and for three responses) pass:

```
$ pytest test_survey.py
======================== test session starts ========================
--snip--
test_survey.py ..                                                [100%]
========================= 2 passed in 0.01s =========================
```

This works perfectly. However, these tests are a bit repetitive, so we'll use another feature of `pytest` to make them more efficient.

Using Fixtures

In *test_survey.py*, we created a new instance of `AnonymousSurvey` in each test function. This is fine in the short example we're working with, but in a real-world project with tens or hundreds of tests, this would be problematic.

In testing, a *fixture* helps set up a test environment. Often, this means creating a resource that's used by more than one test. We create a fixture in `pytest` by writing a function with the decorator `@pytest.fixture`. A *decorator* is a directive placed just before a function definition; Python applies this directive to the function before it runs, to alter how the function code behaves. Don't worry if this sounds complicated; you can start to use decorators from third-party packages before learning to write them yourself.

Let's use a fixture to create a single survey instance that can be used in both test functions in *test_survey.py*:

```
import pytest
from survey import AnonymousSurvey

❶ @pytest.fixture
❷ def language_survey():
    """A survey that will be available to all test functions."""
```

```
    question = "What language did you first learn to speak?"
    language_survey = AnonymousSurvey(question)
    return language_survey

❸ def test_store_single_response(language_survey):
    """Test that a single response is stored properly."""
❹   language_survey.store_response('English')
    assert 'English' in language_survey.responses

❺ def test_store_three_responses(language_survey):
    """Test that three individual responses are stored properly."""
    responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
❻       language_survey.store_response(response)

    for response in responses:
        assert response in language_survey.responses
```

We need to import `pytest` now, because we're using a decorator that's defined in `pytest`. We apply the `@pytest.fixture` decorator ❶ to the new function `language_survey()` ❷. This function builds an `AnonymousSurvey` object and returns the new survey.

Notice that the definitions of both test functions have changed ❸ ❺; each test function now has a parameter called `language_survey`. When a parameter in a test function matches the name of a function with the `@pytest.fixture` decorator, the fixture will be run automatically and the return value will be passed to the test function. In this example, the function `language_survey()` supplies both `test_store_single_response()` and `test_store_three_responses()` with a `language_survey` instance.

There's no new code in either of the test functions, but notice that two lines have been removed from each function ❹ ❻: the line that defined a question and the line that created an `AnonymousSurvey` object.

When we run the test file again, both tests still pass. These tests would be particularly useful when trying to expand `AnonymousSurvey` to handle multiple responses for each person. After modifying the code to accept multiple responses, you could run these tests and make sure you haven't affected the ability to store a single response or a series of individual responses.

The structure above will almost certainly look complicated; it contains some of the most abstract code you've seen so far. You don't need to use fixtures right away; it's better to write tests that have a lot of repetitive code than to write no tests at all. Just know that when you've written enough tests that the repetition is getting in the way, there's a well-established way to deal with the repetition. Also, fixtures in simple examples like this one don't really make the code any shorter or simpler to follow. But in projects with many tests, or in situations where it takes many lines to build a resource that's used in multiple tests, fixtures can drastically improve your test code.

When you want to write a fixture, write a function that generates the resource that's used by multiple test functions. Add the `@pytest.fixture`

decorator to the new function, and add the name of this function as a parameter for each test function that uses this resource. Your tests will be shorter and easier to write and maintain from that point forward.

TRY IT YOURSELF

11-3. Employee: Write a class called `Employee`. The `__init__()` method should take in a first name, a last name, and an annual salary, and store each of these as attributes. Write a method called `give_raise()` that adds $5,000 to the annual salary by default but also accepts a different raise amount.

Write a test file for `Employee` with two test functions, `test_give_default_raise()` and `test_give_custom_raise()`. Write your tests once without using a fixture, and make sure they both pass. Then write a fixture so you don't have to create a new employee instance in each test function. Run the tests again, and make sure both tests still pass.

Summary

In this chapter, you learned to write tests for functions and classes using tools in the `pytest` module. You learned to write test functions that verify specific behaviors your functions and classes should exhibit. You saw how fixtures can be used to efficiently create resources that can be used in multiple test functions in a test file.

Testing is an important topic that many newer programmers aren't exposed to. You don't have to write tests for all the simple projects you try as a new programmer. But as soon as you start to work on projects that involve significant development effort, you should test the critical behaviors of your functions and classes. You'll be more confident that new work on your project won't break the parts that work, and this will give you the freedom to make improvements to your code. If you accidentally break existing functionality, you'll know right away, so you can still fix the problem easily. Responding to a failed test that you ran is much easier than responding to a bug report from an unhappy user.

Other programmers will respect your projects more if you include some initial tests. They'll feel more comfortable experimenting with your code and be more willing to work with you on projects. If you want to contribute to a project that other programmers are working on, you'll be expected to show that your code passes existing tests and you'll usually be expected to write tests for any new behavior you introduce to the project.

Play around with tests to become familiar with the process of testing your code. Write tests for the most critical behaviors of your functions and classes, but don't aim for full coverage in early projects unless you have a specific reason to do so.