LEARNING MADE EASY

2nd Edition

Python®

ALL-IN-ONE

for dummies®
A Wiley Brand

7 Books in one!

John C. Shovic, PhD
Alan Simpson

IN THIS CHAPTER

- Discovering why Python is hot
- Finding the tools for success
- Writing Python in VS Code
- Writing Python in Jupyter notebooks

Chapter 1
Starting with Python

Because you're reading this chapter, you probably realize that Python is a great language to know if you're looking for a good job in programming, or if you want to expand your existing programming skills into exciting cutting-edge technologies such as artificial intelligence (AI), machine learning (ML), data science, or robotics, or even if you're just building apps in general. So we're not going to try to sell you on Python. It sells itself.

Our approach leans heavily toward the hands-on. A common failure in many programming tutorials is that they already assume you're a professional programmer in some language, and they skip over things they assume you already know.

This book is different in that we *don't* assume that you're already programming in Python or some other language. We *do* assume that you can use a computer and understand basics such as files and folders.

We also assume you're not up for settling down in an easy chair in front of the fireplace to read page after page of theoretical stuff about Python, like some kind of boring novel. You don't have that much free time to kill. So we're going to get right into it and focus on *doing,* hands-on, because that's the only way most of us learn. We've never seen anyone read a book about Python and then sit at a computer and write Python like a pro. Human brains don't work that way. We learn through practice and repetition, and that requires being hands-on.

Why Python Is Hot

We promised we weren't going to spend a bunch of time trying to sell you on Python, and that's not our intent here. But we would like to talk briefly about *why* it's so hot.

Python is hot primarily because it has all the right stuff for the kind of software development that's driving the software development world these days. Machine learning, robotics, artificial intelligence, and data science are the leading technologies today and for the foreseeable future. Python is popular mainly because it already has lots of capabilities in these areas, while many older languages lag behind in these technologies.

Just as there are different brands of toothpaste, shampoo, cars, and just about every other product you can buy, there are different brands of programming languages with names such as Java, C, C++ (*pronounced C plus plus*), and C# (*pronounced C sharp*). They're all programming languages, just like all brands of toothpaste are toothpaste. The main reasons cited for Python's current popularity are

- Python is relatively easy to learn.
- Everything you need to learn (and do) in Python is free.
- Python offers more ready-made tools for current hot technologies such as data science, machine learning, artificial intelligence, and robotics than most other languages.

HTML, CSS, AND JavaScript

Some of you may have heard of languages such as HTML, CSS, and JavaScript. Those aren't traditional programming languages for developing apps or other generic software. HTML and CSS are specialized for developing web pages. And although JavaScript is a programming language, it is heavily geared to website development and isn't quite in the same category of general programming languages like Python and Java.

If you specifically want to design and create websites, you have to learn HTML, CSS, and JavaScript whether you're already familiar with Python or some other programming language.

Figure 1-1 shows Google search trends over the last five years. As you can see, Python has been gaining in popularity (as indicated by the upward slope of the trend) whereas other languages have stayed about the same or declined. This certainly supports the notion that Python is the language people want to learn right now and for the future. Most people would agree that given trends in modern computing, learning Python gives you the best opportunity for getting a secure, high-paying job in the world of information technology.

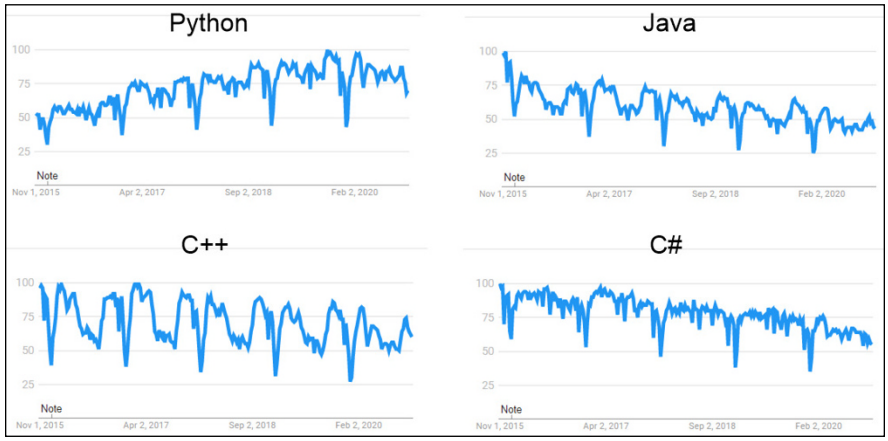


FIGURE 1-1: Google search trends for the last five years or so.

TIP

You can do your own Google trend searches at `https://trends.google.com`.

Choosing the Right Python

There are different *versions* of Python out roaming the world, prompting many a beginner to wonder things such as

- Why are there different versions?
- How are they different?
- Which one should I learn?

All good questions, and we'll start with the first. A version is kind of like a car year. You can go out a buy a 1968 Ford Mustang, a 1990 Ford Mustang, a 2019 Ford Mustang, or a 2020 Ford Mustang. They're all Ford Mustangs. The only difference is that the one with the highest year number is the most current Ford Mustang. That Mustang is different from the older models in that it has some improvements based on experience with earlier models, as well as features current with the times.

Programming languages (and most other software products) work the same way. But as a rule we don't ascribe year numbers to them because they're not released on a yearly basis. They're released whenever they're released. But the principle is the same. The version with the highest number is the newest, most recent model, sporting improvements based on experience with earlier versions, as well as features relevant to the current times.

Just as we use a decimal point with money to separate dollars from cents, we use decimal points with version numbers to indicate how much the software has changed. When there's a significant change, the entire version number is usually changed. More minor changes are expressed as decimal points. You can see how the version number increases along with the year in Table 1-1, which shows the release dates of various Python versions. We've skipped a few releases because there is little reason to know or understand the differences between all the versions. We present the table only so you can see how newer versions have higher version numbers; that's all that matters.

TABLE 1-1 Examples of Python Versions and Release Dates

Version	When Released
Python 3.9	October 2020
Python 3.8	October 2019
Python 3.7	June 2018
Python 3.6	December 2016
Python 3.5	September 2015
Python 3.4	March 2014
Python 3.3	September 2012
Python 3.2	February 2011
Python 3.1	June 2009
Python 3.0	December 2008
Python 2.7	July 2010
Python 2.6	October 2008
Python 2.0	October 2000
Python 1.6	September 2000
Python 1.5	February 1997
Python 1.0	January 1994

If you paid close attention, you may have noticed that Version 3.0 starts in December 2008, but Version 2.7 was released in 2010. So if versions are like car years, why the overlap?

The car years analogy just indicates that the larger the number, the more recent the version. But in Python, the year is the most recent within the main Python version. When the first number changes, that's usually a change that's so significant, software written in prior versions may not even work in that version. If you happen to be a software company with a product written in Python 2 on the market, and have millions of dollars invested in that product, you may not be too thrilled to have to start over from scratch to go with the current version. So older versions often continue to be supported and evolve, independent of the most recent version, to support developers and businesses that are already heavily invested in the previous version.

The biggest question on most beginners minds is "what version should I learn?" The answer to that is simple . . . whatever is the most current version. You'll know what that is because when you go to the Python.org website to download Python, it will tell you the most current stable build (version). That's the one they'll recommend, and that's the one you should use.

The only reason to learn something like Version 2 or 2.7 or something else older would be if you've already been hired to work on some project and the company requires you to learn and use a specific version. That sort of situation is rare, because as a beginner you're not likely to already have a full-time job as a programmer. But in the messy real world, some companies are heavily invested in an earlier version of a product, so when hiring, they'll be looking for people with knowledge of that version.

In this book, we focus on versions of Python that are current in late 2020 and early 2021, from Python 3.9 and above. Don't worry about version differences after the first and second digits. Version 3.9.1 is similar enough to version 3.9.0 that it's not important, especially to a beginner. Likewise, Version 3.9 isn't that big a jump from 3.8. So don't worry about these minor version differences when first learning. Most of what's in Python is the same across all recent versions. So you need not worry about investing time in learning a version that is or will soon be obsolete.

Tools for Success

Now we need to start getting your computer set up so you can learn, and do, Python hands-on. For one, you'll need a good Python interpreter and editor. The *editor* lets you type the code, and the *interpreter* lets you run that code. When you

run (or execute) code, you're telling the computer to "do whatever my code tells you to do."


TECHNICAL STUFF

The term *code* refers to anything written in a programming language to provide instructions to a computer. The term *coding* is often used to describe the act of writing code. A code editor is an app that lets you type code, in much the same way an app such as Word or Pages helps you type regular plain-English text.

Just as there are many brands of toothpaste, soap, and shampoo in the world, there are many brands of code editors that work well with Python. There isn't a right one or a wrong one, a good one or a bad one, a best one or a worst one. Just a lot of different products that basically do the same thing but vary slightly in their approach and what that editor's creators think is good.

If you've already started learning Python and are happy with whatever you've been using, you're welcome to continue using that and ignore our suggestions. If you're just getting started with this stuff, we suggest you use VS Code, because it's an excellent, free learning environment.

Introducing Anaconda and VS Code

The editor we recommend and will be using in this book is called Visual Studio Code, officially. But most often, it is spoken or written as *VS Code*. The main reasons why it's our favorite follow:

- It is an excellent editor for learning coding.
- It is an excellent editor for writing code professionally and is used by millions of professional programmers and developers.
- It's relatively easy to learn and use.
- It works pretty much the same on Windows, Mac, and Linux.
- It's free.

The editor is an important part of learning and writing Python code. But you also need the Python interpreter. Chances are, you're also going to want some Python packages. Packages are simply code written by someone else to do common tasks so that you don't have to start from scratch and reinvent the wheel every time you want to perform one of those tasks.


TIP

Python packages are not a crutch for beginners. They are major components of the entire Python development environment and are used by seasoned professionals as much as by beginners.

Historically, managing Python, the packages, and the editor was a somewhat laborious task involving typing cryptic commands at a command prompt. Although that's not a particularly bad thing, it isn't the most efficient way to do things, especially when you're first getting started. You end up spending most of your time upfront trying to learn and type awkward commands just to get Python to work on your computer, rather than learning Python itself.

An excellent alternative to the old command-line driven ways of doing things is to use a more complete Python development environment with a more intuitive and easily managed graphic user interface, as on a Mac or Windows or any phone or tablet. The one we recommend is Anaconda. It is free and excellent. If you've never heard of it and aren't so sure about downloading something you've never heard of, you can explore what it's all about at the Anaconda website at `www.anaconda.com`.

Anaconda is often referred to as a data-science platform because many of the packages that come with it are data-science oriented. But don't let that worry you if you're interested in doing other things with Python. Anaconda is excellent for learning and doing all kinds of things with Python. And it also comes with VS Code, our personal favorite coding editor, as well as Jupyter Notebook, which provides another excellent means of coding with Python. And best of all, it's 100 percent free, so it's well worth the effort of downloading and installing it.

We can't take you step-by-step through every part of downloading and installing Anaconda because it's distributed from the website, and people change their websites whenever they feel like it. But we can certainly give you the broad strokes. You should be able to follow along using Mac, Windows, or Linux. Just keep an eye on the screen as you go along, and follow any onscreen instructions as they arise, while following the steps here.

Installing Anaconda and VS Code

To download and install Anaconda, and VS Code, you'll need to connect to the Internet and use a web browser. Any web browser should do, be it Chrome, Firefox, Safari, Edge, Internet Explorer, or whatever. Fire up whatever browser you normally use to browse the web, then follow these steps:

1. **Browse to `www.anaconda.com/download` to get to their download page.**

   Don't worry about version numbers or dates.

2. **Keep scrolling down or click a Download button, and you should find options that look something like the example shown in Figure 1-2.**

   We can't say exactly what the page will look like the day you visit. We used a Windows computer for that screenshot, but Mac and Linux users will see something similar.

FIGURE 1-2: Click Download under the largest version number.

3. **Click Download under whichever version number is the highest on your screen.**

   The highest number for us was version 3.8, but a higher-numbered version may be available when you get there. Don't worry about that.

TIP

Jot down the Python version number you're downloading for future reference a little later in this chapter. You can also click How to Install ANACONDA (or however the link might be worded when you get there) on the download page if you'd like to see the instructions from the Anaconda team.

4. **Follow any onscreen instructions to download the free version.**

   If you see information about becoming a commercial user (where you have to pay money), follow the onscreen instructions to download the free version. You'll have to set up a user account.

5. **When the download is complete, open your Downloads folder (or wherever you downloaded the file).**

6. **If you're using Mac or Linux, double-click the file you downloaded. If you're using Windows, right-click that file and choose Run as Administrator, as shown in Figure 1-3.**

TECHNICAL STUFF

The Run-As-Administrator business in Windows ensures that you can install everything. If that option isn't available to you, double-clicking the file's icon should be sufficient.

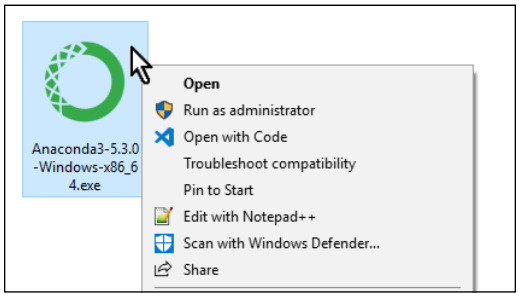


FIGURE 1-3: In Windows, right-click and choose Run As Administrator.

7. **Click Next, Continue, Agree, or I Agree on the first installation pages until you get to one of the pages shown in Figure 1-4.**

   Mac is the one on the left, and Windows is on the right.

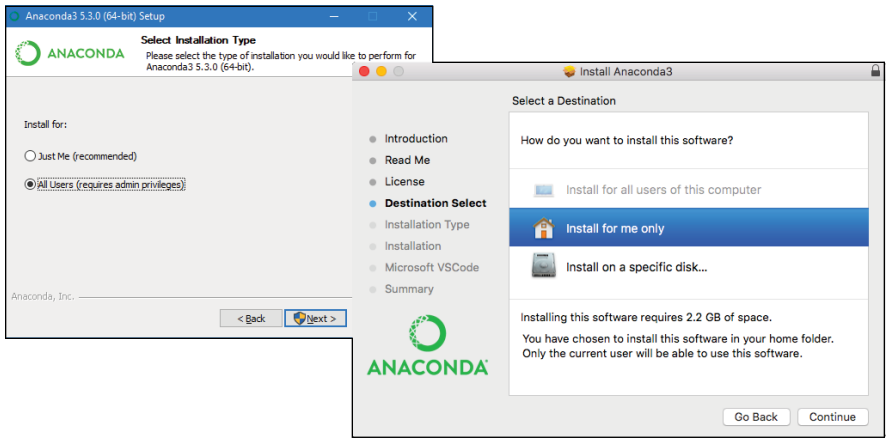


FIGURE 1-4: Choose how to install Anaconda.

8. **Choose whichever option makes sense to you.**

   If in doubt, Mac users can choose Install on a specific disk and then Macintosh HD. Windows users with Administrator privileges can choose Install for All Users. If the option we suggested isn't available to you, click the one closest to it.

9. **Click Continue or Next and follow the onscreen instructions.**

   If you're unsure about what options to choose on any page, don't choose any option. Just accept the default suggestions.

10. **When you come to a page where it asks if you want to install Microsoft VS Code (it may take quite a while), click Install Microsoft VS Code (or whatever option on your screen indicates that you want to install VS Code).**

TIP

   If VS Code is already installed on your computer, no worries. The Anaconda installer will just tell you that, or perhaps update your version to the more current version.

11. **Continue to follow any onscreen instructions, clicking Continue or Next to proceed through the installation steps, and then click Close or Finish on the last page.**

You may be prompted to sign up with Anaconda Cloud. Doing so is free but not required. Decide for yourself if that's something you want to do.

Opening Anaconda (Mac)

After Anaconda is installed on your Mac, you can open it as you would any other app. Use whichever of the following methods appeals to you:

- Open Launch Pad and click the Anaconda Navigator icon to open it.
- Click the Spotlight magnifying glass, start typing **Anaconda**, and then double-click Anaconda Navigator.
- Open Finder and your Applications folder and double-click the Anaconda Navigator icon.

After Anaconda Navigator opens, right-click its icon in the dock and choose Keep in Dock. That way, its icon will be visible in the dock at all times and easy to find.

Opening Anaconda (Windows)

After Anaconda is installed in Windows, you can start it as you would any other app. Although there are some differences among different versions of Windows, you should be able to use either of these two options:

- Click the Start button, and then click Anaconda Navigator on the Start menu.
- Click the Start button, start typing **Anaconda**, and then click Anaconda Navigator on the Start menu when you see it there.

On the Start menu, you can right-click Anaconda Navigator and choose Pin to Start or right-click and choose More➪Pin to Taskbar to make the icon easy to find in the future.

Using Anaconda Navigator

Anaconda Navigator, as the name implies, is the component of the Anaconda environment that lets you navigate around through different features of the app and choose what you want to run. When you first start Navigator, it opens to the Anaconda Navigator home page, which should look something like Figure 1-5.


TIP

If you see a prompt to get an updated version when you open Anaconda, it's okay to install the update. It won't cost anything or affect your ability to follow along in this book.

The left side of the Anaconda Navigator home page has options such as Home, Environments, Learning, and Community. They're not directly related to learning and doing Python, so you're welcome to explore them on your own.

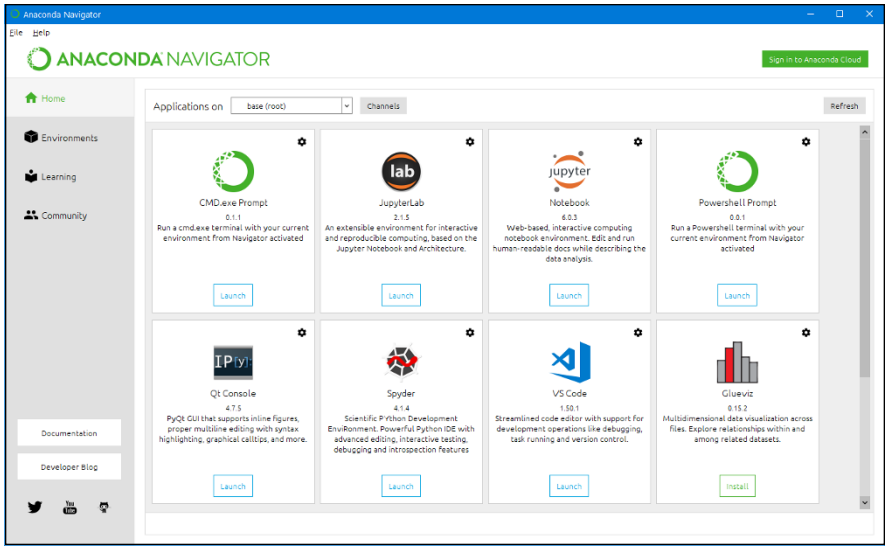


FIGURE 1-5: Anaconda Navigator home page.

Writing Python in VS Code

Most of the Python coding we do here, we'll do in VS Code. Whenever you want to use VS Code to write Python, we suggest that you open VS Code from Anaconda Navigator rather than from the Start menu or Launch Pad. That way, VS Code will already be pointing to the version of Python that comes with Anaconda, which is easier than trying to figure out all that yourself. So the steps are

1. **If you haven't already done so, open Anaconda Navigator.**
2. **Scroll down a little until you see the Launch button under VS Code, if necessary, and then click the Launch button.**

ABOUT GIT

Git is a way to store backups of your coding projects and share coding projects with other developers or team members. It's popular with professional programmers, and VS Code has built-in support for it. But Git is optional and not directly related to learning or doing Python coding, so it's perfectly okay to choose Don't Show Again to bypass that offer when it arrives. You can install Git at any time if you later decide to learn about it.

The first time you open VS Code, you may be prompted to make some decisions. None of them are required, so you can just click the X in the upper-right corner of the each one. However, the one that mentions Git will keep popping up unless you click Don't Show Again.

When you're finished, the VS Code window will look something like Figure 1-6. If you don't see quite that many options on your screen, choose Help➪Welcome from the menu bar.

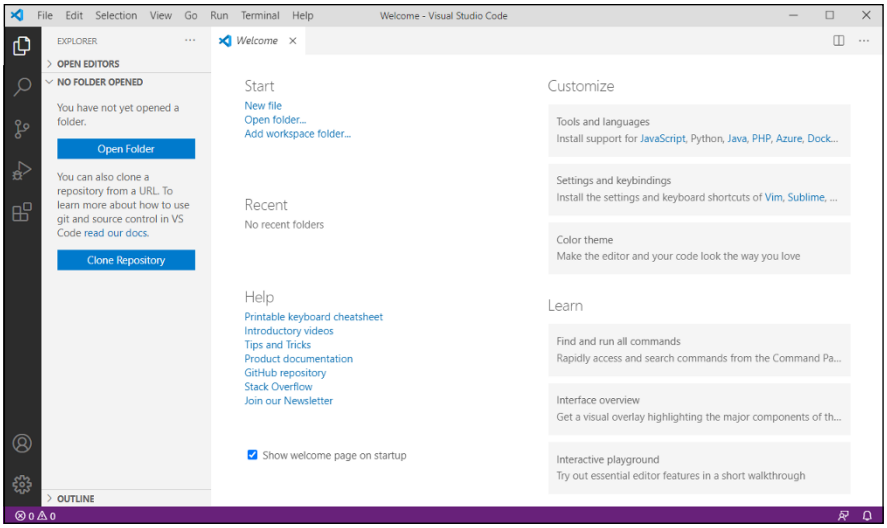


FIGURE 1-6: The welcome screen of VS Code editor.

Your screen will likely be black with white and colored text. In this book, we show everything as white with black text because it's easier to read on paper that way. You can keep the dark background if you like. If you would rather have a light background, choose Code➪Preferences➪Color Theme (Mac) or File➪Preferences➪Color Theme (Windows). Then choose a lighter color theme; if you choose Light (Visual Studio), your VS Code screens will look more like the ones in this book.

Visual Studio Code is a generic code editor that works with many different languages. To use VS Code with Python and Anaconda, you need some VS Code *extensions*. But you should already have them because they come with your Anaconda download. To verify that, click the Extensions icon in the left pane (it looks like a puzzle piece). You should see at least three extensions listed: Anaconda Extension Pack, Python, and YAML, as shown in Figure 1-7.

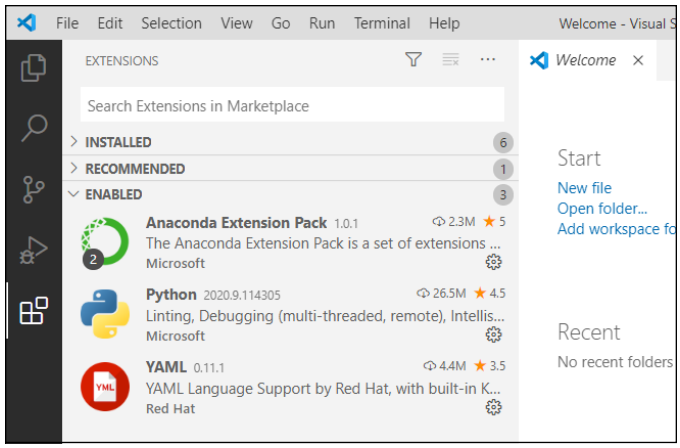


FIGURE 1-7: VS Code extensions for Python.

Choosing your Python interpreter

Before you start doing any Python coding in VS Code, you want to make sure you're using the correct Python interpreter. To do so, follow these steps:

1. **Choose View➪Command Palette from VS Code's menu.**
2. **Type** python **and then click Python: Select Interpreter.**

   Choose the Python version number that matches your download (the one you jotted down while first downloading Anaconda). If you have multiple options with the same version number, choose the one that includes the names *base* and *conda*, as in Figure 1-8.

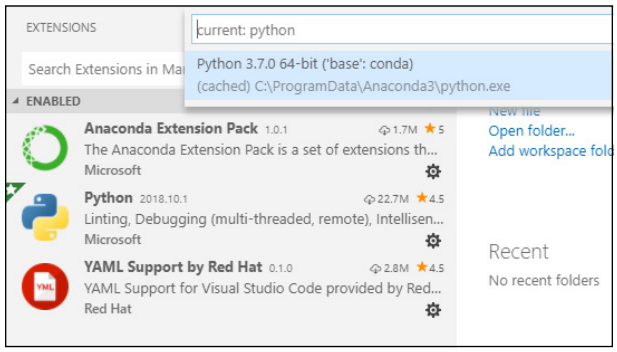


FIGURE 1-8: Choose your Python interpreter (usually the highest version number).

Writing some Python code

To ensure that you'll be able to follow along with the examples in this book, let's make sure VS Code is ready for Python coding. Follow these steps:

1. **In VS Code, choose View➪Terminal from the VS Code menu.**

   You should see a pane along the bottom-right that looks like one of those shown in Figure 1-9.

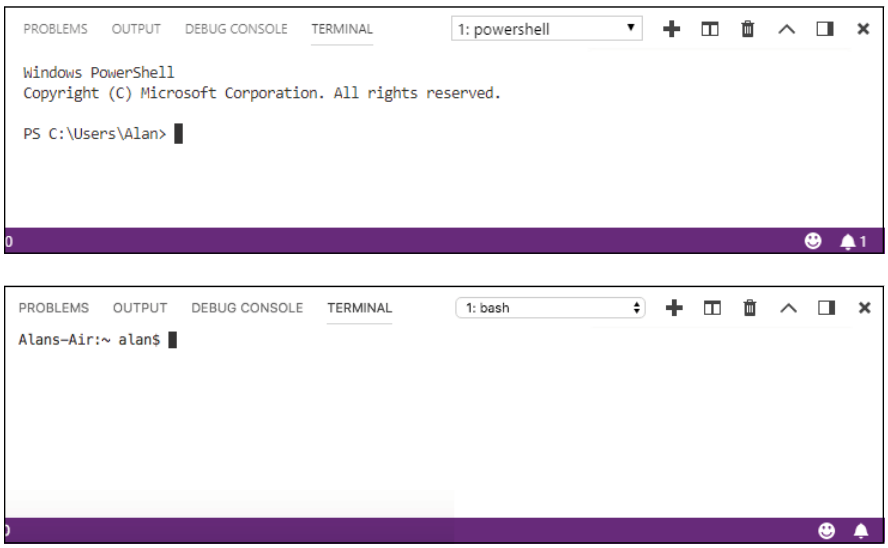


FIGURE 1-9: Terminal in VS Code (Windows and Mac).

2. **In Terminal, type** python **and press Enter.**

   You should see some information about Python followed by a >>> prompt. That >>> prompt is your Python interpreter; if you type Python code there and press Enter, the code will execute.

3. **Type** 1+1 **and press Enter.**

   You should now see 2 (the sum of 1 plus 1), followed by another Python prompt, as shown in Figure 1-10.

The 1+1 exercise is about as simple an exercise as you can do. However, all we care about right now is that you saw 2, because that means your Python development environment is all set up and ready to go. You won't have to repeat any of these steps in the future.

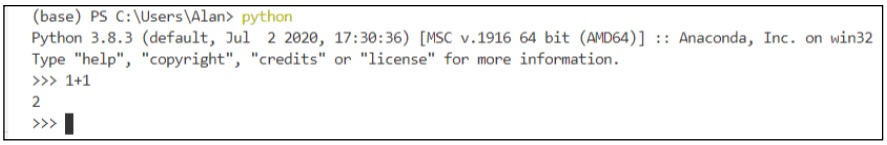


FIGURE 1-10: Python shows the sum of one plus one.

Now we'll show you how to exit Python and VS Code:

1. **In the VS Code Terminal pane, press CTRL+D or type** exit() **and press Enter.**

   The last prompt at the bottom of the Terminal window should now be whatever it was before you went to the Python prompt, indicating that you're no longer in the Interpreter.

2. **To close VS Code:**

   - *Windows: Click the Close icon (X) in the upper-right corner or choose View➪Exit from the menu.*
   - *Mac: Click the round red dot in the upper-left corner, or choose Code➪Quit Visual Studio Code from the menu.*

3. **Close Anaconda Navigator using a similar technique:**

   - *Window: Click the X in the upper-right corner or choose File➪Quit from the menu bar.*
   - *Mac: Click the red dot or go to Anaconda Navigator in the menu and choose Quit Anaconda Navigator.*

Getting back to VS Code Python

In the future, any time you want to work in Python in VS Code, we suggest that you open Anaconda Navigator and then Launch VS Code from there. You'll be ready to roll and do any of the hands-on exercises presented in future chapters.

Using Jupyter Notebook for Coding

Jupyter Notebook is another popular tool for writing Python code. The name *Jupyter* comes from the fact that it supports writing code in three popular languages: **Ju**lia and **Pyt**hon and **R**. Julia and R are popular for data science. Python is a more generic programming language that happens to be popular in data science as well, though Python is good for all kinds of development, not just data science. The *Notebook* part of the name comes from the fact that your code is placed in structures similar to a regular paper notebook.

People often use Jupyter to share code on the Internet. It is free and comes with Anaconda. So if you've installed Anaconda, you already have it and can open it any time by following these simple steps:

1. **Open Anaconda as discussed previously**
2. **Click Launch under Jupyter Notebook, as shown in Figure 1-11.**

   Jupyter notebooks are web-based, meaning that when Jupyter opens, it does so in your default web browser, such as Safari, Chrome, Edge, Firefox, or Internet Explorer. At first, it doesn't look like it has much to do with coding, because it just shows an alphabetized list of folder (directory) names to which it has access, as shown in Figure 1-12. (Of course, the names you see may be different from those in the figure, because those folder names are from our computer, not yours.)

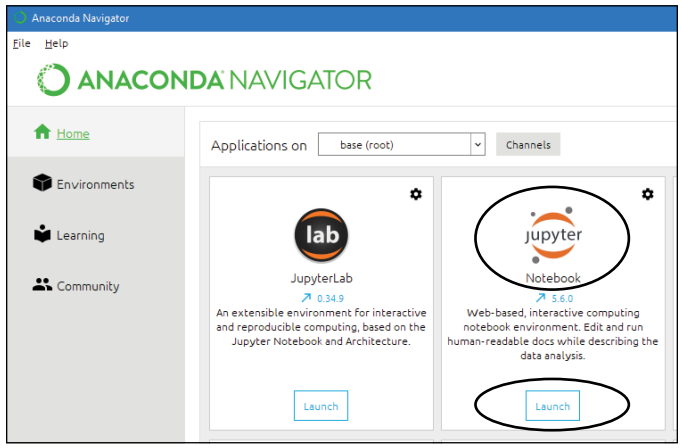


FIGURE 1-11: Launch Jupyter Notebook from Anaconda's home page.

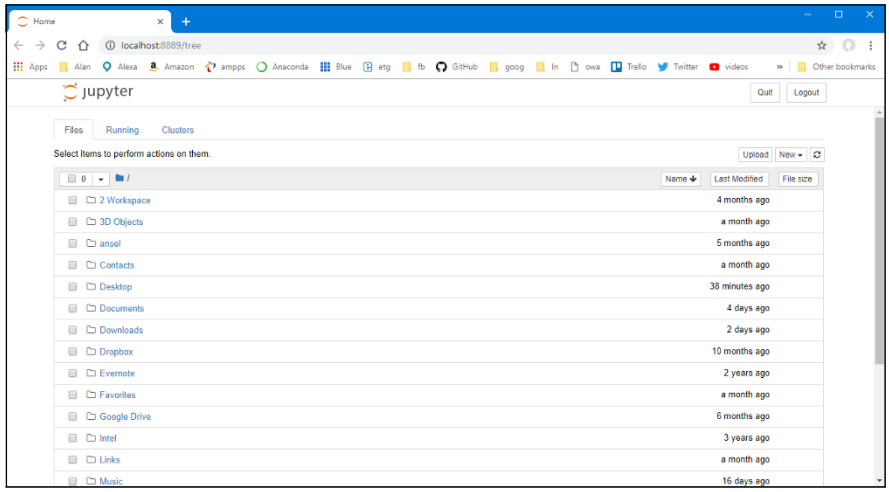


FIGURE 1-12: Jupyter Notebook opening page.

3. **Click a folder name of your choosing (the Desktop is fine; we're not making any commitment here).**

4. **Click New, and then choose Python 3 under Notebook, as shown in Figure 1-13.**

   A new, empty notebook named Untitled opens. You should see a rectangle with `In []:` on the left side. That's called a *cell*, and a cell can contain either code (words written in the Python language) or just regular text and pictures. If you want to write code, make sure the drop-down menu in the toolbar displays Code. Change that menu option to Markdown if you want to write regular text rather than Python code.

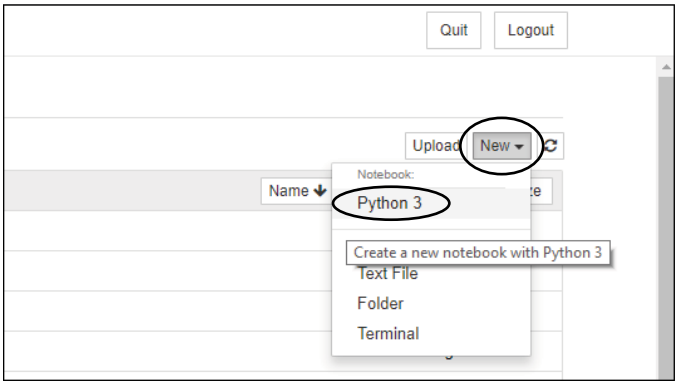


FIGURE 1-13: Creating a new Jupyter notebook.

TECHNICAL STUFF

Markdown is a language for writing text that uses fonts, pictures, and such. We'll talk more about that in the next chapter. For now, let's stay focused on Python code, because that's what this book is all about.

A cell is not like the Python interpreter, where your code executes immediately. You have to type some code first (any amount), and then run that code by clicking the Run button in the toolbar. To see for yourself, follow these steps:

1. **Click inside the code cell.**

2. **Type** `1+1`**.**

3. **Press Enter.**

You see `1+1` in the cell, but not the result, `2`. To get the result, click Run in the toolbar or put the mouse pointer into the cell and click the Run icon to the left of the cell, as shown in Figure 1-14, or click Run in the toolbar above the cell. You'll see the number `2` to the right of `Out[1]`. `Out` indicates that you're seeing the output from executing the code in the cell, which of course is `2` because `1` plus `1` is `2`.

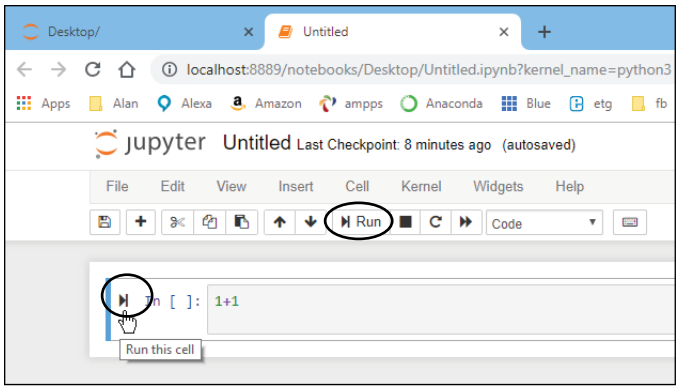


FIGURE 1-14: Two ways to run code in a Jupyter cell.

To close a notebook, do either of these following:

- Close the tab in the browser that's showing the cell.
- Choose File➪Close and Halt from the toolbar above the cells.

Figure 1-15 shows an example using Chrome as the browser. Your tabs may look different if you're using a different browser.

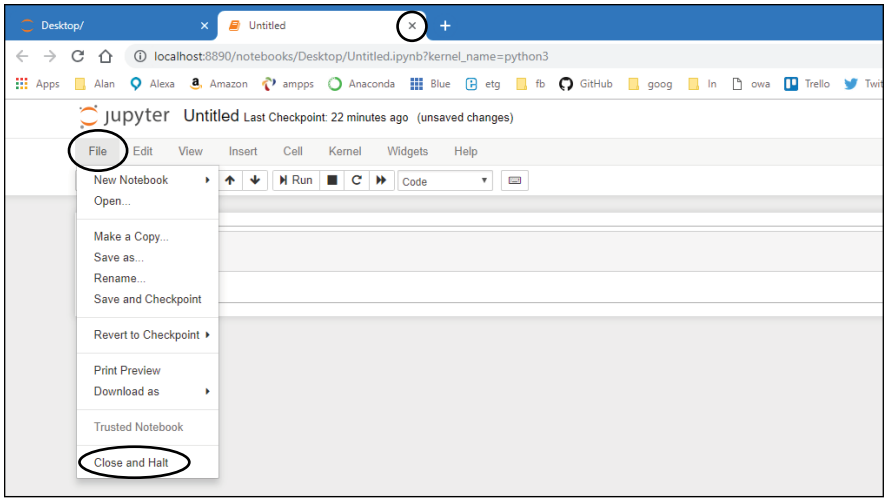


FIGURE 1-15: Result of running code in a Jupyter Notebook cell.

You may be prompted to save your work. For now, you don't need to save because we're focused on the absolute basics . . . what you'll do every time you run Python code.

Even if you don't specifically save a notebook, you'll see an icon for it in the folder in which you created the notebook. The notebook's name will be Untitled, and if you have filename extensions visible, you'll see the .ipynb filename extension. The *pynb* part is short for Python notebook. The *i* in that extension, in case you're wondering, comes from iPython, which is the name of the app from which Jupyter Notebook was created and is short for interactive.

You can delete a notebook file if you're just practicing and don't want to keep it. Just make sure you close the notebook in the web browser (or just close the browser first) — otherwise, you may get an error message stating that you can't delete the file while it's open.

So now you're ready to go. You have a great set of tools set up for learning Python. The simple skills you've learned in this chapter will serve you well through your learning process, as well as your professional programming after you've mastered the basics. Come on over to Chapter 2 in this minibook now and we'll delve a bit deeper into Python and using the tools you now have available on your computer.

IN THIS CHAPTER

- Using interactive mode
- Creating a development workspace
- Creating a folder for your code
- Typing, editing, and debugging code
- Writing code in a Jupyter

Chapter 2

Interactive Mode, Getting Help, and Writing Apps

Now that you've installed Anaconda and VS Code, you're ready to start digging deeper into writing Python code. In this chapter, we take you briefly through the interactive, help, and code-editing features of VS Code and Jupyter Notebook to build on what you've learned so far. Most of you are probably anxious to get started on more advanced topics such as data science, artificial intelligence, robotics, or whatever. But learning those topics will be easier if you have a good understanding of the many tools available to you — and the skills to use them.

Using Python's Interactive Mode

Many teachers and authors will suggest that you try things hands-on at the Python prompt, and assume you already know how to get there. We've seen many frustrated beginners complain that trying activities recommended in some tutorial never work for them. The frustration often stems from the fact that they're

typing and executing the code in the wrong place. With Anaconda, the Terminal pane in VS Code is a great place to type Python code. So in this chapter that's where you'll start.

Opening Terminal

To use Python interactively with Anaconda, follow these steps:

1. **Open Anaconda Navigator, and then open VS Code by clicking its Launch button on the Anaconda home page.**
2. **If you don't see the Terminal pane at the bottom of the VS Code window, choose View➪Terminal from the VS Code menu bar.**
3. **If the word *Terminal* isn't highlighted at the top of the pane, click Terminal (circled in Figure 2-1).**

FIGURE 2-1: The Terminal pane in VS Code.

The first prompt you see is typically for your computer's operating system, and likely shows the user name of the account you're using. For example, on a Mac, it may look like `Alans-Air:~ alan$` but with the name of your computer in place of `Alans-Air`. In Windows it would likely be `C:\Users\Alan>`, with your user name in place of `Alan`, and possibly a different path than `C:\Users`.

For example, on a Mac, we see this prompt:

```
Alans-Air:~ alan$
```

And in Windows, we see this:

```
C:\Users\Alan>
```

COLORS AND ICONS IN VS CODE

By default, the VS Code Terminal pane displays white text against a black background. We reverse those colors in this book because dark text against a light background is easier to see in a printed book. You can use any color scheme you like. If you want to switch to black on white, as shown in this book, choose File (Windows) or Code (Mac) and then choose Preferences ➪ Color Theme ➪ Light (Visual Studio).

If you want your icons in VS Code to match the ones we use, you'll need to download and install the Material Icon theme. You may also want to download the Material Color theme and try it out; we don't use it for the book because it doesn't play well when printed on paper. Follow these steps:

1. **Click the Extensions icon (puzzle piece) in the left pane.**
2. **Type** material**, look for Material Icon Theme, and click its Install option.**
3. **If you see a prompt at the bottom right asking if you want to activate the icons, click Activate.**
4. **Choose File (Windows) or Code (Mac), choose Preferences ➪ File Icon Theme, and then click Material Icon Theme.**

   If you don't see the Material icon as an option, make sure you've downloaded the extension.
5. **If you'd like to try out the Material color theme, open File (in Windows) or Code (on a Mac), choose Preferences ➪ Color Theme, and then click Material Icon Theme.**

If at any time you change your mind about the color theme, repeat Step 5 and choose something other than Material Icon Theme.

Depending on your Windows version and current configuration, you might see the following prompt instead, where `xxx` is your user name:

```
PS C:\Users\xxx>.
```

This just means that you're using PowerShell. You don't need to change anything. The command shown here will work with PowerShell too.

You would see your user name in place of `Alan` and possibly a different path than `C:\Users`.

Getting your Python version

At the operating system command prompt, type the following and press Enter to see what version of Python you're using. Note the space before the first hyphen, and no other spaces.

```
python --version
```

You should see something like Python 3.*x.x* (where the *x*'s are numbers representing the version of Python you're using). If instead you see an error message, you're not quite where you need to be. You want to make sure you start VS Code from Anaconda, not just from Launch Pad or your Start menu. Type **python --version** in the VS Code Terminal pane, and press Enter again. If it still doesn't work, choose View➪Command Palette from the VS Code menu bar, type **python**, choose Python: Select Interpreter, and then choose the Python interpreter you downloaded with Anaconda.

Going into the Python Interpreter

When you're able to enter `python --version` and not get an error, you're ready to work with Python in VS Code. From there you can get into the Python interpreter by entering the command

```
python
```

REMEMBER

When we, or anyone else, says "enter the command," that means you have to type the command and then press Enter. Nothing happens until you press Enter. So if you just type the command and wait for something to happen, you'll be waiting for a long, long time.

A NOTE ABOUT PyLint

PyLint is a feature of Anacaona that helps you find and avoid errors in your code. It's usually turned on by default. The first time you try to use Python, you might see some messages in the lower-right corner of VS Code. If you see a message about *Python Language Server,* click Try It Now and then click Reload. If you see a message that *Linter PyLint Is Not Installed,* click Install.

If you see *Select Python Environment* near the lower-left corner of VS Code's window, click that and choose the Anaconda option from the menu that drops down near the top center. If you see multiple Anaconda options, choose the one with the largest version number.

You should see some information about the Python version you're using and the >>> prompt, which represents the Python interpreter.

Entering commands

Entering commands in the Python interpreter is the same as typing them anywhere else. You must type the command correctly, and then press Enter. If you spell something wrong in the command, you will likely see an error message, which is just the interpreter telling you it doesn't understand what you mean. But don't worry, you can't break anything. For example, suppose you type the command

```
howdy
```

After you press Enter, you'll see some techie gibberish that is trying to tell you that the interpreter doesn't know what "howdy" means, so it can't do that. Nothing has broken. You're just back to another >>> prompt, where you can try again, as shown in Figure 2-2.

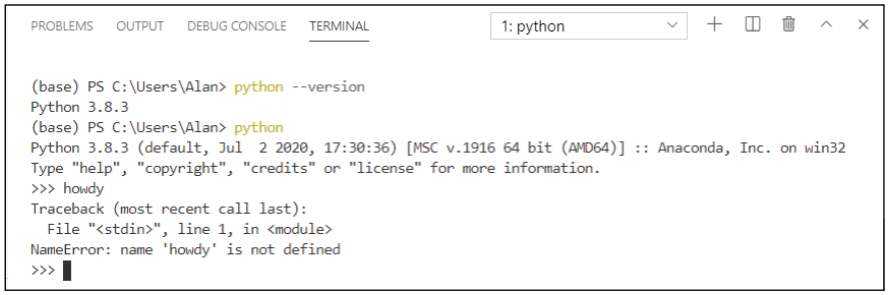


FIGURE 2-2: Python doesn't know what *howdy* means.

Using Python's built-in help

One of the prompts in Figure 2-2 mentions that you can type `help` as a command in the Python interpreter. Note that you don't type the quotation marks, just the word *help* (and then press Enter, as always). This time you see

```
Type help() for interactive help, or help(object) for help about object.
```

Now the interpreter is telling you to type **help** followed by an empty pair of parentheses, or help with a specific word in parentheses (*object* is the example given). Even though you're told to type the command, you should type it and press Enter. Go ahead and enter the following:

```
help()
```

Note that the line does not have spaces. After you press Enter, the screen provides some information about using Python's interactive help, something like the example shown in Figure 2-3.

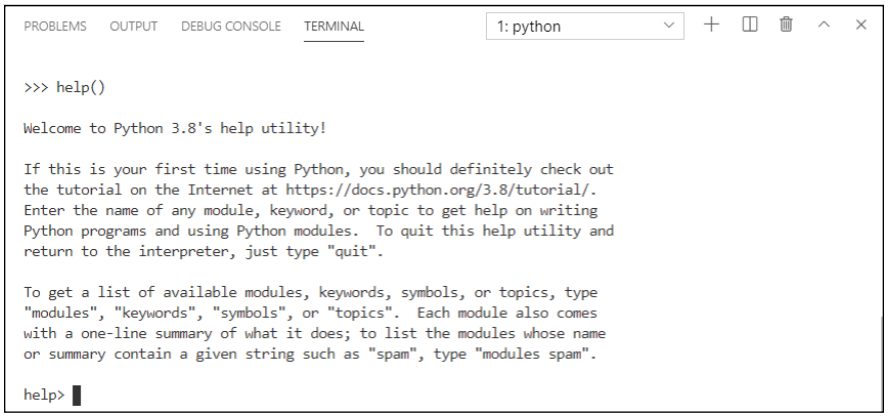


FIGURE 2-3: Python's interactive help utility.

Seeing `help>` at the bottom of the window tells you that you're no longer in the operating system shell or the Python interpreter (which always shows `>>>`) but are now in a new area that provides help. As described on the screen, you can enter the name of any module, keyword, or topic to get help with that term. As a beginner, you might not need help with specifics right at the moment. But it's good to know that the help is there if you need it.

For example, Python uses certain keywords, which have special meaning in the language. To get a list of those, just type the following at the `help>` prompt:

```
keywords
```

After you press Enter, you'll see a list of keywords, as shown in Figure 2-4.

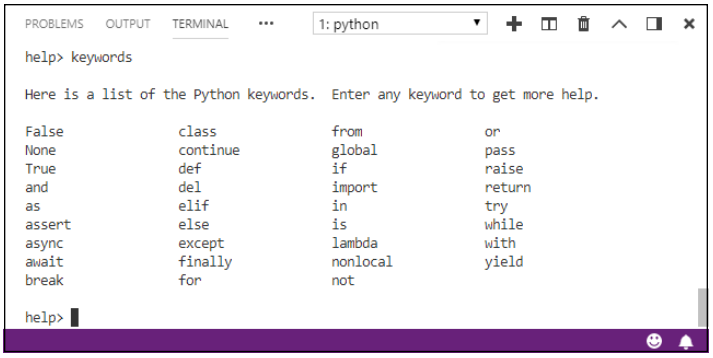


FIGURE 2-4: Keyword help.

Above the list of keywords is a message telling you that you can type any keyword at the `help>` prompt for more information about that keyword. For example, entering the `class` keyword provides information about Python classes, as shown in Figure 2-5. These are not the kind of classes you attend at school; rather, they're the kind you create in Python (after you've learned the basics and are ready to move onto more advanced topics).

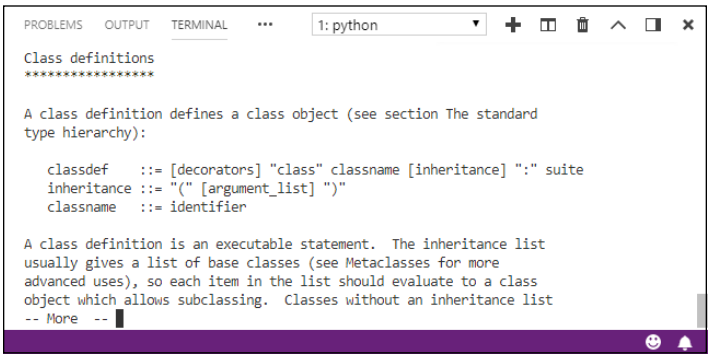


FIGURE 2-5: Python class help.

All the technical jargon in the help text is going to leave the average beginner flummoxed. But as you learn about new concepts in Python, realize that you can use the interactive help for guidance as needed.

The `--More--` at the bottom of the text isn't a prompt where you type commands. Instead, it just lets you know that there is more text, perhaps several pages worth. Press the spacebar or Enter to see it. Every time you see `-- More --`, you can press the spacebar or Enter to get to the next page. Eventually you'll get back to the `help>` prompt. If you want to quit rather than keep scrolling, press the letter q.

Exiting interactive help

To get out of interactive help and return to the Python prompt, type the letter **q** (for quit) or press Ctrl+Z. You should be back at the `>>>` prompt. At the `>>>` prompt, type `exit()` or `python`.

To leave the Python prompt and get back to the operating system, type **exit()** and press Enter. Note that if you make a mistake, such as forgetting the parentheses, you'll get some help on the screen. For example, if you type `exit` and press Enter, you'll see

```
Use exit() or Ctrl-Z plus Return to exit.
```

You'll know you've exited the Python interpreter when you see the operating system prompt rather than >>> at the end of the Terminal window, as in Figure 2-6.

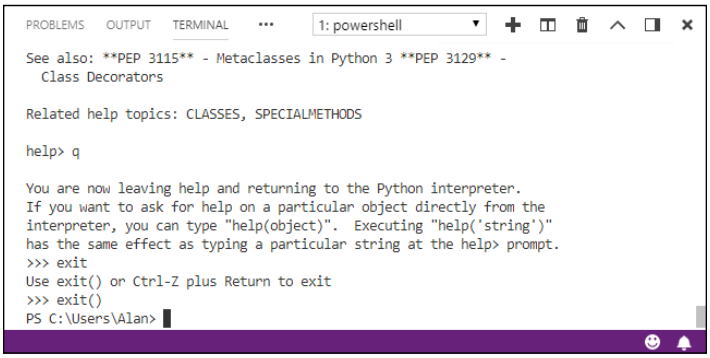


FIGURE 2-6: Back to the operating system prompt.

Searching for specific help topics online

Python's built-in help is somewhat archaic because it's text oriented rather than interactive, but it can help you when you need a quick reminder about some Python keyword you've forgotten. But if you're online, you're better off searching the web for help. If you're looking for videos, start at `www.youtube.com`; if not, `https://stackoverflow.com/` is a good place to ask questions and search for help. And of course there's always Google, Bing, and other search engines.

Regardless of what you use to search, remember to start your search with the word *python* or *python 3*. A lot of programming languages share similar concepts and keywords, so if you don't specify the Python language in your search request, there's no telling what kinds of results you may get.

Lots of free cheat sheets

Other good resources for learners are the countless cheat sheets available online for free. Whenever you start to feel overwhelmed by all the possibilities of a language like Python, a cheat sheet summarizing things to a single page or so can help bring information to a more manageable (and less intimidating) size.

Of course, you're not really cheating with a cheat sheet, unless you use it while taking a test that you're supposed to answer from memory. But writing code in real life is much different from answering multiple-choice questions. So what we often call a *cheat sheet* in the tech world is just another tool to help us learn. Many types of cheat sheets are available — what appeals to you depends on your learning style. To see what's available, head to Google or Bing or any search engine you

like and search for *free python 3 cheat sheet*. Most are in a format you can download, print, and keep handy as you learn the seemingly infinite possibilities of writing code in Python.

Creating a Python Development Workspace

Although interactive modes and online help are decent support tools, most people want to use Python to create apps. We've found that creating apps is easiest if you set up a VS Code development environment specifically for learning and coding Python. You can set up other development environments for coding in other languages, such as HTML, CSS, and JavaScript for the web, fine-tuning each as you go along to best support whatever language you're working in.

We often switch between Mac and Windows computers, so we have one development environment for each. Alan keeps his in a OneDrive folder so he can get to them from anywhere. Although this is not a requirement, it's handy. If you'll be working strictly from one computer, however, you can put your environment on your computer's hard drive rather than on a cloud drive.

VS Code uses the term *workspace* to define what we call a *development environment.* That environment is the Python interpreter you're using plus any additional extensions you gather along the way.

You can store your workspaces anyplace you like. Do so now, before proceeding with the next set of steps:

1. **If VS Code isn't already open, launch it from Anaconda.**
2. **Choose File➪Save Workspace As, and navigate to the folder where you want to save the workspace settings.**
3. **Type a name for the workspace, and then click Save.**

   In Figure 2-7, Alan is saving his workspace in Windows (top) and then on a Mac (bottom).
4. **Next, do one of the following, depending on whether you're using a Mac or Windows, to adjust some VS Code settings to indicate the location of that saved workspace:**

   - On a Mac, choose Code➪Preferences➪Settings.
   - In Windows, choose File➪Preferences➪Settings.

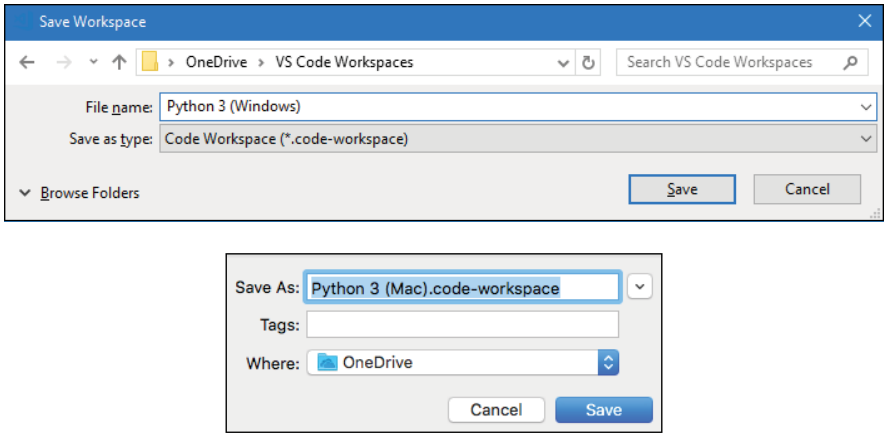


FIGURE 2-7: Saving current settings as workspace settings.

5. **If you see a page like the one in Figure 2-8, click the Open Settings (JSON) icon near the top-right corner (and circled in the figure).**

   If you don't see that icon, look for one that allows you to open Settings.json.

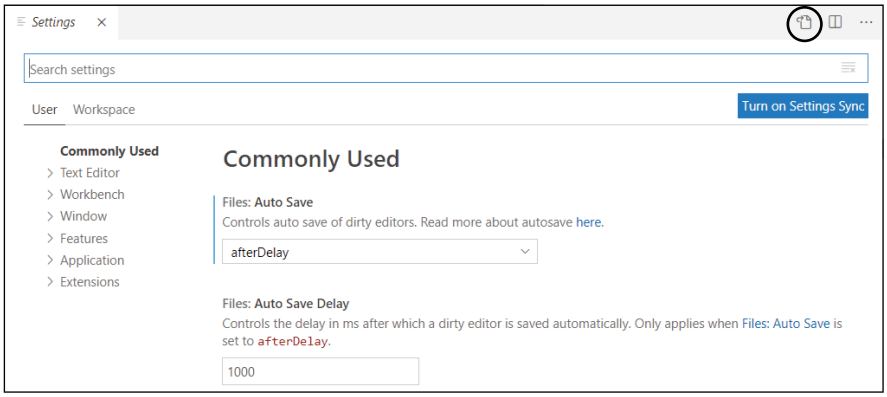


FIGURE 2-8: VS Code Settings.

6. **In the next window, select the entire line of code that starts with `python.pythonpath` (that line tells VS Code where to find the Python interpreter on your computer).**

   You can also select any other command lines that you'd like to make part of the workspace, but don't select the curly braces.

7. **Right-click the selected code and choose Copy to copy it to the clipboard.**

8. **Click the Split Editor Right icon, near the top-right corner.**

   (The icon looks like two side-by-side pages in the version of VS code we're using right now.) Two copies of settings.json appear side-by-side on the screen.

9. **Choose View➪Command Palette. Type** open **and then select Preferences: Open Workspace Settings (JSON).**

10. **Click between the setting's curly braces and paste the lines of code there, as shown in Figure 2-9.**

    All the settings in settings.json (on the top) are copied to the Python 3 settings (on the bottom).

FIGURE 2-9: Python path copied to Workspace Settings.

11. **Choose File➪Save from the VS Code menu.**

12. **Close the Settings and User Settings tabs by clicking the X on the right side of each tab.**

13. **Close VS Code, and then close Anaconda.**

You'll see how to take advantage of the new workspace settings in a moment.

Creating a Folder for Your Python Code

Next, you create a folder to store all the Python code that you write in this book, so it's all together in one place and easy to find when you need it. You can put this folder anywhere you like and name it whatever you like.

In Windows you can navigate to the folder that will contain the new folder (Alan uses OneDrive, but you can use Desktop, Documents, or any other folder). Right-click an empty place in the folder. Then choose New Folder (Mac) or New➪Folder (Windows). Type the folder name and press Enter. To follow along with the examples in this chapter, name your folder *AIO Python.*

Now you should associate this code folder with the VS Code workspace you just created, so that any time you work in the folder you're using the correct Python interpreter and other Python-related settings you choose over time with the files in the code folder. Here's how:

1. **Open Anaconda and launch VS Code from there.**
2. **From the VS Code menu, choose File➪Open Workspace.**
3. **Navigate to the folder where you saved your workspace and open the workspace from there.**
4. **Choose File➪Add Folder to Workspace.**
5. **Navigate to the folder in which you created the folder for your Python code, click that folder's icon, and choose Add.**

The Explorer bar in VS Code, shown in Figure 2-10, indicates that you've opened both the workspace — Python 3 (Windows) Workspace in the figure — and, under that workspace, the code folder — AIO Python in the figure. If you see something entirely different in the left pane, click the Explorer icon, at the top-left corner of the VS Code window, to make sure you're viewing the Explorer pane.

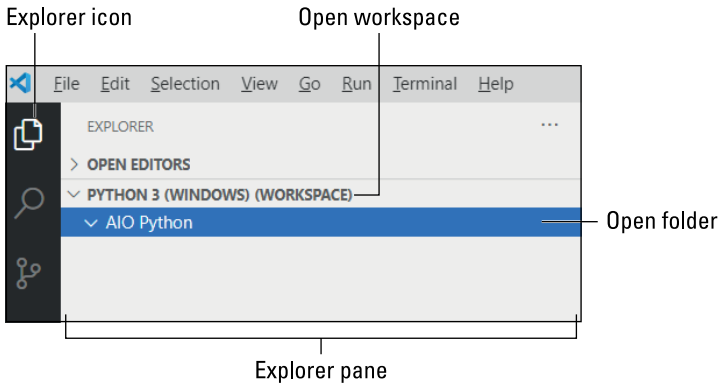


FIGURE 2-10: Python 3 workspace and AIO Python folder open in VS Code.

When you expand the Open Editors bar, near the top of the Explorer pane, you see files that are currently open in VS Code. Each open file is represented by a tab across the top of the editing area to the right. Right now, in the image, no files were open. But if, say, the VS Code Welcome page is open right now on your own screen, you see Welcome on the right. To close that page, click the X next to its name in the Explorer pane or on the tab. Any time you want to reopen the Welcome page, choose Help➪Welcome from the VS Code menu bar.

TIP

If you see a symbol other than a triangle, or no symbol at all, before a folder name, you maybe be using an icon theme that's different from the default. No worries, just click to the left of any folder to expand or collapse it.

You went through quite a few steps to set up your workspace. The benefit, especially if you use VS Code to work in multiple languages, is that any time you want to work with Python in VS Code, all you have to do is follow these steps:

1. **If you've closed VS Code, launch it from Anaconda Navigator.**
2. **Choose File➪Open Workspace from the VS Code menu.**
3. **Open your workspace.**

   The workspace and any folders you've associated with that workspace open, and you're ready to go.

Typing, Editing, and Debugging Python Code

Most likely, you'll write the vast majority of code in an editor. As you probably know, an *editor* enables you to type and edit text. Code is text. The editor in VS Code is set up for typing and editing code, so you may hear it referred to as a *code editor*.

Each Python code file you create will be a plain text file with a `.py` filename extension. We suggest that you keep any files you create for this book in that AIO Python folder, which you should be able to see anytime VS Code and your Python 3 workspace are open.

To create a `.py` file at any time, follow these steps:

1. **If you haven't already done so, open VS Code and your Python 3 workspace.**
2. **If the Explorer pane isn't open, click the Explorer icon near the top-left of VS Code.**
3. **To create a file in your AIO Python folder, right-click the folder name and choose New File, as shown in Figure 2-11.**

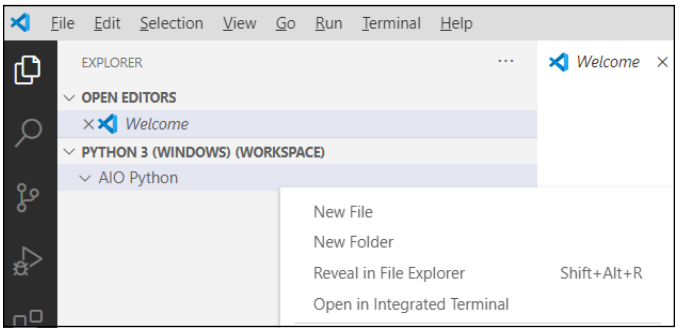


FIGURE 2-11: Right-click a folder name and choose New File.

4. **Type the filename with the `.py` extension (**hello.py **for this first one) and press Enter.**

   The new file opens and you can see its name in the tab on the right, as shown in Figure 2-12. The larger area below the tab is the editor, where you type the Python code. The filename also appears under the AIO Python folder name in the Explorer pane, because that's where it's stored. You can click the Open Editors line to expand and collapse it, which just reveals the names of documents that are currently open for editing.

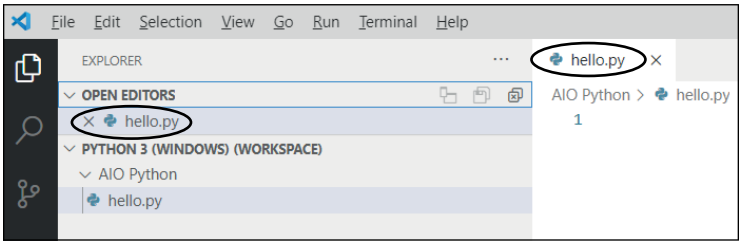


FIGURE 2-12: New `hello.py` file is open for editing in VS Code.

Writing Python code

Now that you have a `.py` file open, you can use it to write some Python code. As is typical when learning a new programming language, you'll start by typing a simple Hello World program. Here are the steps:

1. **Click just to the right of line 1 in the editing area.**
2. **Type the following:**

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

As you're typing, you may notice text appearing on the screen. That text is *IntelliSense text,* which detects what you're typing and shows you some information about that keyword. You don't have to do anything with that, though — just keep typing.

3. **Press Enter after you've typed the line.**

The new line of code is displayed on the screen. You may also notice a few other changes, as shown in Figure 2-13:

- The Explorer icon sports a circled 1, indicating that you currently have one unsaved change.
- The `hello.py` name in the tab and the same filename under Open Editors (if that section is expanded) displays a dot, to indicate that the file has unsaved changes.

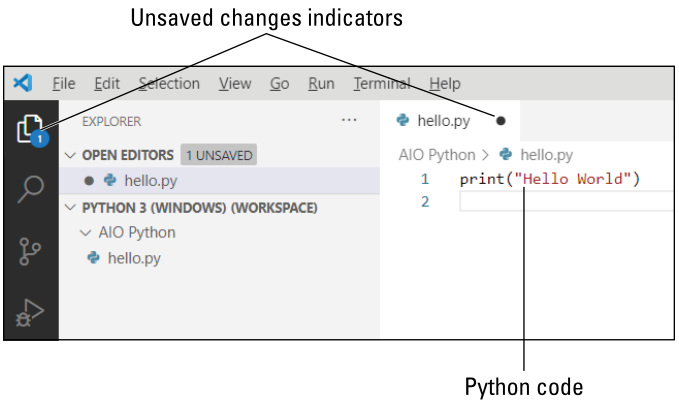


FIGURE 2-13: The `hello.py` file contains some Python code and has unsaved changes.

Saving your code

Code you type in VS Code is not saved automatically. There are two ways to deal with that. One is to try to remember to save any time you make a change that's worth saving. The easiest way to do that is to choose File➪Save from VS Code's menu bar or press Ctrl+S in Windows or ⌘+S on a Mac.

We prefer the second method, which is to use AutoSave to automatically save changes we make. To enable Auto Save, choose File➪Auto Save from VS Code's menu bar. The check mark next to Auto Save means that it's turned on. To turn off AutoSave, just choose File➪Auto Save again. The file is saved automatically as you make changes.

Running Python in VS Code

To test your Python code in VS Code, you need to run it. The easiest way to do that is to right-click the file's name (`hello.py` in this example) and choose Run Python File in Terminal, as shown in Figure 2-14.

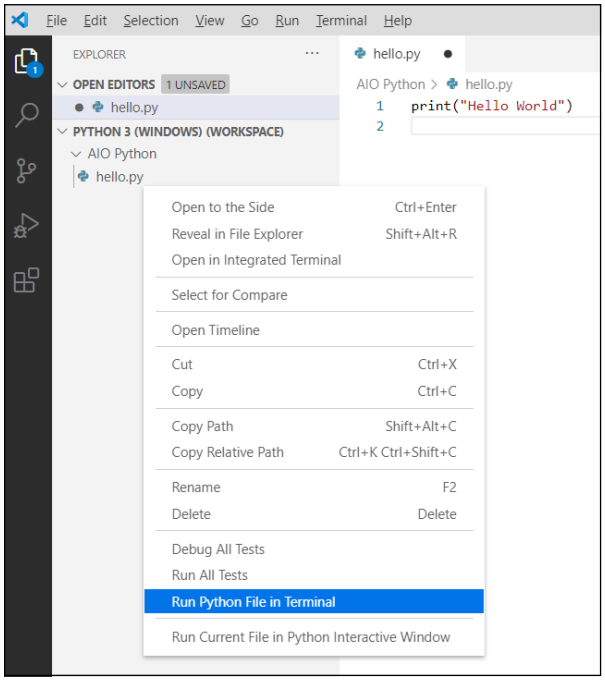


FIGURE 2-14: Run `hello.py`.

The Terminal pane opens along the bottom of the VS Code window. You'll see a command prompt followed by a comment to run the code in the Python interpreter (`python.exe`). And below that, you'll see the output of the program: the words *Hello World*, in this example, and then another prompt, as shown in Figure 2-15. This app is not the most exciting one in the world, but at least now you know how to write, save, and execute a Python program in VS Code, a skill you'll be using often as you continue through this book and through your Python programming career.

FIGURE 2-15: Output from `hello.py`.

If you're using PowerShell in the Terminal window, you may see a message about switching to the command prompt. Unless you happen to be a PowerShell expert and need it (for whatever reason), you might as well click Use Command Prompt if you see that option so the prompt won't keep pestering you.

Learning simple debugging

When you're first learning to write code, you're bound to make a lot of mistakes. Realize that mistakes are no big deal — you won't break or destroy anything. The code just won't work as expected.

Before you attempt to run some code, you might see several screen indications of an error in your code:

- The name of the folder and file that contain the error will be red in the Explorer pane.
- The number of errors in the file will appear in red next to the filename in the Explorer bar.
- The total number of errors will appear next to the circled X in the bottom left corner of the VS Code window.
- The bad code will have a wavy red underline.

In Figure 2-16, we typed `PRINT` in all uppercase, which is not allowed in Python. Python is case-sensitive and the correct command is `print`. Remember, when we show a command to type in lowercase, you have to type it in lowercase, too.

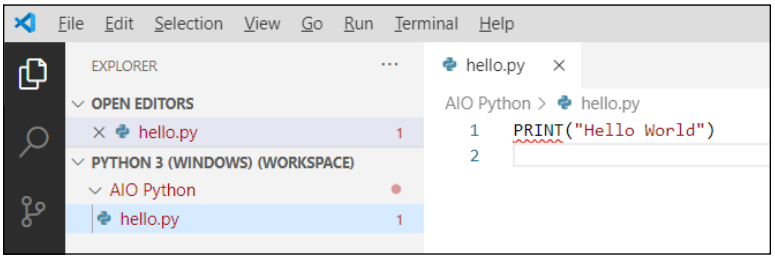


FIGURE 2-16: `PRINT` is typed incorrectly in `hello.py`.

To run the file in Terminal, you must fix the error. Hover the mouse pointer over the word with the red wavy underline to see a brief (and highly technical) description of the problem. In the example shown in Figure 2-16, we would just replace *PRINT* with *print,* and then save the change (unless we've turned on Auto Save). Then we can right-click and choose Run Python File in Terminal to run the corrected code.

Using the VS Code Python debugger

VS Code has a built-in debugger that helps when working with more complex programs and provides a means of testing Python programs in VS Code. You won't be writing anything super complex right now. But there's no harm in getting the debugger set up and ready as part of your Python development workspace. Follow these steps to do so now:

1. **Click the debug icon to the left of the Explorer pane.**

   (The Debug icon is a right-facing triangle with a bug in the corner.) The Debug pane opens.

2. **If you see Create a Launch.json File link, click it.**

3. **Click the drop-down menu at the top of the pane and choose Workspace, and then click Python File Debug the Currently Active Python File, as shown in Figure 2-17.**

4. **Close launch.json by clicking the X on its tab.**

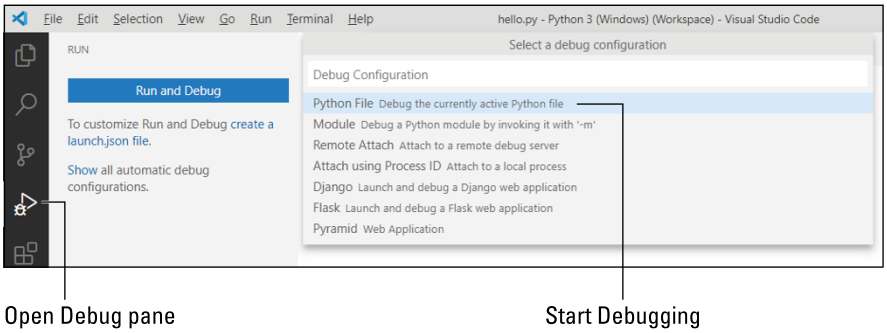


FIGURE 2-17: VS Code Debug pane.

From now on, as an alternative to using the right-click method to run Python code, you can use the Debug pane. The debugger always works with the *current file*, which is whatever file is selected (highlighted) in the Explorer pane. So the usual steps for using the debugger will likely be as follows:

1. **Click the Explorer pane to see a list of all your files.**

2. **Click the icon or file name of the file you want to debug.**

   At the moment you have only one file, `hello.py`. The `hello.py` filename is highlighted in the Explorer pane. The highlight tells you that `hello.py` is the current file that the debugger will run when you tell it to.

3. **Open the Debug pane again by clicking the Debug icon in the left bar of VS Code again.**

4. **Click the Start Debugging icon.**

   The icon is a green triangle next to Python: Current File (workspace).

When you click the Start Debugging icon, the Python code will run as it did when you chose Run Python File in Terminal. If your code has an error, you'll get additional help on the screen describing the error.

If that seems like a lot to remember, for now all you have to remember is that whenever you want to run some Python code in VS Code, you can do either of the following:

- Right-click the `.py` file's name and choose Run File in Terminal.
- Click the `.py` file's name in the Explorer bar to select the file, click the Debug icon, and then click the Start Debugging icon, at the top of the Debug pane. Optionally, you can click Run and choose Start Debugging, or press the F5 key.

If you can remember those two options for running Python files, you're well on your way to learning Python.

We're going to look at a different way to write Python code next. So feel free to close any files you have open as well as VS Code.

Writing Code in a Jupyter Notebook

In Chapter 1 of this minibook, you learned that you can write and run Python code in a Jupyter notebook. In this section, we show you how to create, save, and open a Jupyter notebook. For our example, we create a subfolder named *Jupyter Notebooks* inside the AIO Python folder. You can, of course, save your Jupyter notebook wherever you want using any filenames you want.

Creating a folder for Jupyter Notebook

A Jupyter Notebooks folder is no different from any other folder, so you can create it using whatever method you normally use in your operating system. We put ours

in the AIO Python folder we created, again just to keep all the files for this book in one place:

1. **Open your AIO Python folder (or whatever folder you created for working with files in this book) in Finder (Mac) or Explorer (Windows).**
2. **Right-click an empty spot in that folder, and choose New➪Folder (Windows) or New Folder (Mac).**
3. **Type** Jupyter Notebooks **as the folder name and press Enter.**

Now that you have a folder in which to save Jupyter notebooks, you can create a notebook, as discussed next.

Creating and saving a Jupyter notebook

To create a Jupyter notebook and save it in a folder, follow these steps:

1. **Open Anaconda (if it isn't already open) and launch Jupyter Notebooks.**
2. **Navigate to the Jupyter Notebooks folder you created in the preceding section.**

   Because the folder is empty, you should see a message stating that the notebook list is empty.
3. **Click New and choose Python 3.**
4. **Near the top of the new notebook that opened, click Untitled, type** 01 Notebook **as the new name, and click Rename.**

   See Figure 2-18. The notebook is created and saved as 01 Notebook.

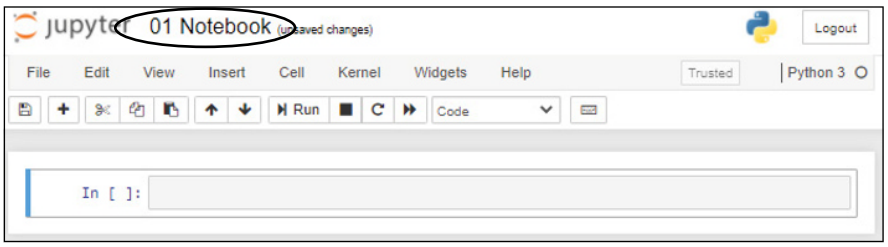


FIGURE 2-18: A saved Jupyter notebook.

Below the menu bar and toolbar in the notebook you'll see a large rectangular box next to `In[]:`. That box is a typing area called a *cell.* Next, you'll type some code in that cell.

Typing and running code in a notebook

When your notebook is open, you see at least one cell. When you see Code in the drop-down menu in the toolbar below the menu bar, the active cell is for typing code. To take Jupyter Notebook for a spin, follow these steps:

1. **Click in the Code cell to the right of `In [ ]:` and type the following:**

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

   Don't forget to use lowercase letters for the word *print*.

2. **To run the code, hold down the Alt key (Windows) or Option key (Mac) and press Enter, or click the Run button in the toolbar above the code.**

   The output from the code appears below the cell.

Adding Markdown text

As mentioned, you can add text (and pictures and video) to Jupyter notebooks. When typing regular text, you don't need to use any special coding. If you want to format the text or add pictures or videos, however, you'll need to use Markdown tags. *Markdown* is a popular markup language, something similar to a greatly simplified HTML.

We can't go into a lengthy tutorial on Markdown here, and you don't need it to write Python code. And Markdown is easy enough to learn just by searching *Markdown tutorials* in your favorite search engine or on YouTube. But for those who know Markdown already, or are just curious, we'll take you through the steps for creating a cell that contains Markdown text in Jupyter Notebook.

First, make sure you're in your Jupyter Notebook app. If you don't have an empty cell under your Python code, choose Insert➪Insert Cell Below. Then click in that new cell to add content to it. Type your text and Markdown in the cell.

We used the text and Markdown code shown in Figure 2-19.


TIP

If you're interested in learning more about Markdown, check out Alan's free video tutorials in his online school at `https://alansimpson.thinkific.com/courses/easy-markdown-with-vs-code`.

To run a cell that contains Markdown, click the cell and then click Run in the toolbar. The code is rendered into text and any other content you've put in the cell, as shown in Figure 2-20.

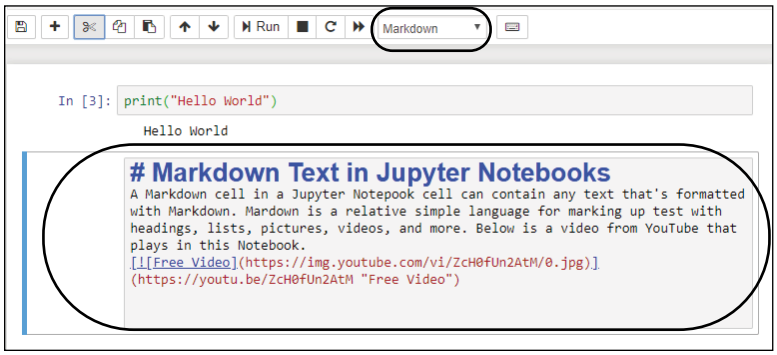


FIGURE 2-19: A Markdown cell containing some Markdown content.

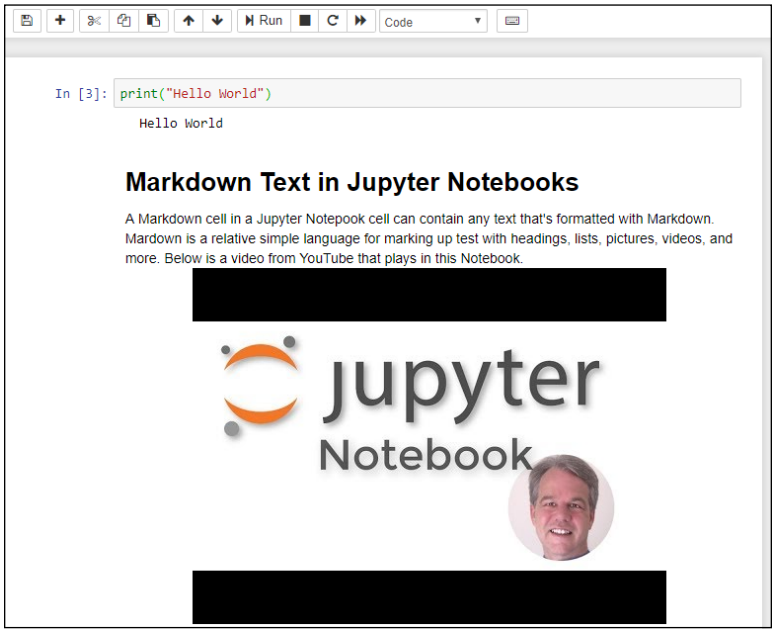


FIGURE 2-20: A Markdown cell with some Markdown code and text in it.

To change code in a code cell, just click the cell and type your code normally. To change the content of a Markdown cell, first double-click some text or the empty space inside the cell so you can see the code again, and then make your changes.

With either type of cell, click Run again after making your changes. Note that only the cell that contains the cursor will run again. If you want to run all the cells in a notebook, use the double triangle icon in the toolbar. It's just to the left of the icon that lets you choose between a code cell and a Markdown cell.

Saving and opening notebooks

To save a Jupyter notebook, choose File➪Save and Checkpoint from the menu. Optionally, you can click the little Save and Checkpoint icon (a floppy disk) on the left side of the toolbar.

To close a notebook, choose File➪Close and Halt from the menu.

Any time that you want to reopen a notebook, open Anaconda and launch Jupyter Notebook. Then navigate to the file you saved and click its filename. The filename will probably have the `.ipynb` filename extension, which is standard for Jupyter notebooks.

It's worth noting that when you open the folder (the one we named AIO Python), you'll see the new Jupyter Notebooks subfolder inside that folder. When you open that subfolder, each notebook will be in there as its own file with the `.ipynb` filename extension.

Okay, so you've dug a little deeper in VS Code and Jupyter Notebook, mostly so you can save and open Python files and Jupyter notebooks. These skills will prove useful when you start getting deeper into writing Python code. See you there!

IN THIS CHAPTER

- Understanding the Zen of Python
- Introducing object-oriented programming
- Discovering why indentations are important
- Using Python modules

Chapter 3
Python Elements and Syntax

Many programming languages focus on things that the computer does and how it does them rather than on the way humans think and work. This one simple fact makes most programming languages difficult for most people to learn. Python, however, is based on the philosophy that a programming language should be geared more toward how humans think, work, and communicate than what happens inside the computer. The Zen of Python is the perfect example of that human orientation, so we start this chapter with that topic.

The Zen of Python

The *Zen of Python*, shown in Figure 3-1, is a list of the guiding principles for the design of the Python language. These principles are hidden in an *Easter egg*, which is a term for something in a programming language or an app that's not easy to find and that's an inside joke to people who have learned enough of the language or app to be able to find the Easter egg. To get to the Easter egg, follow these steps:

1. **Launch VS Code from Anaconda Navigator and open your Python 3 workspace.**
2. **If the Terminal pane isn't open, choose View ➪ Terminal from the VS Code menu bar.**

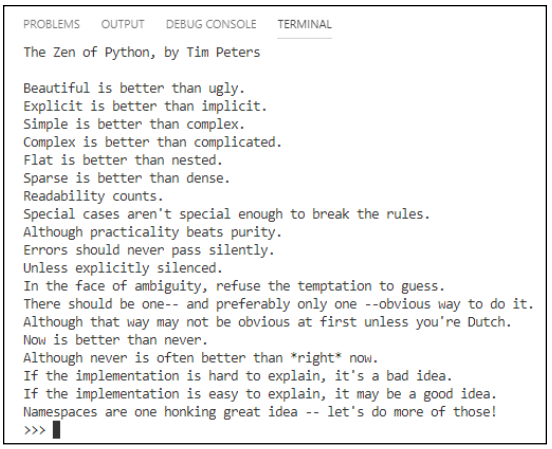


FIGURE 3-1: The Zen of Python.

3. **Type** python **and press Enter to get to the Python prompt (>>>).**

WARNING

If you get an error message after you enter the `python` command, don't panic. You just need to remind VS Code which Python interpreter you're using. Choose View ➪ Command Palette from the menu, type **python**, click Python: Select Interpreter, and choose the Python 3 version that came with Anaconda.

4. **Type** import this **and press Enter.**

The list of aphorisms appears. You may have to scroll up and down or make the Terminal pane taller to see them all. The aphorisms are somewhat tongue-and-cheek in their philosophical rhetoric, but the general idea they express is to always try to make the code more human-readable than machine-readable.

The Zen of Python is sometimes referred to as *PEP 20*, where *PEP* is an acronym for *Python enhancement proposals*. The *20* perhaps refers to the 20 Zen of Python principles, only 19 of which have been written down. We all get to wonder about or make up our own final principle.

Many other PEPs exist, and you can find them all on the Python.org website at `www.python.org/dev/peps`. The one you're likely to hear about the most is PEP 8, which is the *Style Guide for Python Code*. The guiding principle for PEP 8 is "readability counts" — that is, readable by *humans*. Admittedly, when you're first learning Python code, most other peoples' code will seem like some gibberish scribbled down by aliens, and you may not have any idea what it means or does. But as you gain experience with the language, the style consistency will become more apparent, and you'll find it easier and easier to read and understand other peoples' code, which is an excellent way to learn coding yourself.

We'll fill you in on Python coding style throughout the book. Trying to read about it before working on it is sure to bore you to tears. So for now, any time you hear mention of PEP, or especially PEP 8, remember that it's a reference to the Python Coding Style Guidelines from the Python.org website, and you can find it any time you like by doing a web search for *pep 8*. PEP 8 is also referred to as Pycodestyle, especially in VS Code.

This PEP 8 business can be a double-edged sword for learners. On one hand, you don't want to learn a bunch of bad habits only to discover later that you have to unlearn them. On the other hand, the strict formatting demands of PEP 8 can frustrate many learners who are just trying to get their code to work.

To ward off this potential frustration, we follow and explain PEP 8 conventions as we go along. You can take it a step further, if you like, by configuring PyLint to help you. *PyLint* is a tool in Anaconda that makes suggestions about your code as you're typing. (The program adds a little wavy underline near code that may be wrong, and some people think the lines look like lint, hence the name PyLint.)

You can follow these steps to turn on PyLint and PEP 8 now if you'd like to take it for a spin:

1. **Choose File (Windows) or Code (Mac), and then choose Preferences ➪ Settings.**
2. **Under Search Settings, click Workspace.**
3. **In Search Settings, type** pylint **and select the Python > Linting: Enabled option, as shown in Figure 3-2.**
4. **Type** pycodestyle **in the Search box.**
5. **Scroll down and select the Python > Linting: Pycodestyle Enabled option, as in the bottom half of Figure 3-2.**
6. **Choose File ➪ Save All or just Save (if Save All isn't available). Then close the Settings page by clicking the X in its tab.**

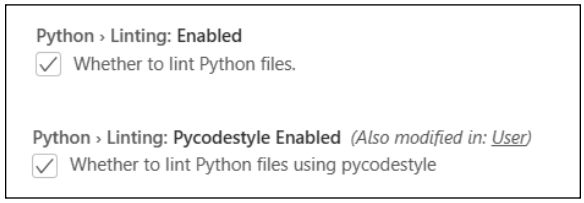


FIGURE 3-2: Workspace settings with PyLint and Pycodestyle (PEP 8) enabled.

All the settings you choose in VS Code are stored in a `settings.json` file. You can make changes to settings also via that file. To get to the file through the Preferences options in VS Code, first choose File (Windows) or Code (Mac) and then choose Preferences➪Settings. Click the Open Settings (JSON) icon near the top right. The icon looks like a paper document with the top corner folded down, and a rounded arrow near the top.

If the Pycodestyle linting is too demanding, you can change the first line of code to `true` and the second line to `false`. (See Figure 3-3.) You can turn off linting altogether by setting the third code line to `false` as well. There are no right or wrong settings, so try what we have for a while and see how it works for you. If you make any changes to `Settings.json`, don't forget to save them and close the settings.json tab.

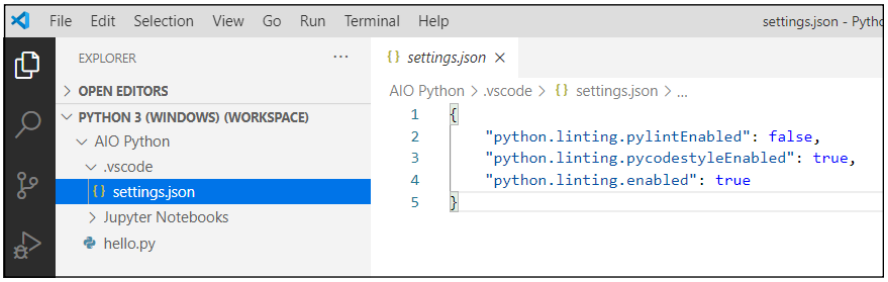


FIGURE 3-3: A different view of Workspace settings.

Introducing Object-Oriented Programming

At the risk of getting too technical or computer science-y, we should mention that there are different approaches to designing languages. Perhaps the most successful and widely used model is *object-oriented programming,* or OOP, which is a design philosophy that tries to mimic the real world in the sense that it consists of objects with properties as well as methods (actions) that those objects perform.

Take a car, for example. Any one car is an object. Not all cars are exactly the same. Different cars have different properties, such as make, model, year, color, and size, which make them different from one another. And yet, they all serve the same basic purpose: to get us from point A to point B without having to walk or use some other mode of transportation.

All cars have certain methods (things they can do) in common. You can drive them, steer them, speed them up, slow them down, control the inside temperature, and more by using controls in the car that you can manipulate with your hands.

An *object* in an object-oriented programming language isn't a physical thing, like a car, because it exists only inside a computer. An object is strictly a software thing. In Python, you can have a *class* (which you can think of as an object creator, such as a car factory) that can produce many different kinds of objects (cars) for varying purposes (sporty, off-road, sedan). All these objects can be manipulated through the controls they all have in common, much as all cars are manipulated by controls such as the steering wheel, brakes, accelerator, and gearshift.

Python is very much an object-oriented language. The core language consists of controls (in the form of words) that allow you to control all different kinds of objects — in your own and other peoples' programs. However, you need to learn the core language first so that when you're ready to start using other peoples' objects, you know how to do so. Similarly, after you know how to drive one car, you pretty much know how to drive them all. You don't have to worry about renting a car only to discover that the accelerator is on the roof, the steering wheel on the floor, and you have to use voice commands rather than a brake to slow it down. The basic skill of driving applies to all cars.

Discovering Why Indentations Count, Big Time

In terms of the basic style of writing code, the one feature that really makes Python different from other languages is that it uses indentations rather than parentheses and curly braces and such to indicate blocks, or chunks, of code. We don't assume that you're familiar with other languages, so don't worry if that statement means nothing to you. But if you are familiar with a language such as JavaScript, you know that you have to do quite a bit of wrangling with parentheses and such to control what's inside of what.

For example, here's some JavaScript code. If you're familiar with the Magic 8 Ball toy, you may have a sense of what this program is doing. But that's not what's important. Just note all those parentheses, curly braces, and semicolons:

```
document.addEventListener("DOMContentLoaded", function () {var question =
   prompt("Ask magic 8 ball a question");var answer = Math.floor(Math.random() *
   8) + 1; if (answer == 1) {alert("It is certain");} else if (answer == 2)
   {alert("Outlook good");} else if (answer == 3) {alert("You may rely on it");}
   else if (answer == 4) {alert("Ask again later");} else if (answer == 5)
   {alert("Concentrate and ask again");} else if (answer == 6) {alert ("Reply
   hazy, try again");} else if (answer == 7) {alert("My reply is no");} else if
   (answer == 8) {alert("My sources say no")} else {alert ("That's not a
   question");}alert("The end");})
```

The code is a mess and not fun to read. We can make reading it a little easier by breaking it into multiple lines and indenting some of those lines. (Note that doing so isn't required in JavaScript.) Following is the reformatted code:

```
document.addEventListener("DOMContentLoaded", function () {
    var question = prompt("Ask magic 8 ball a question");
    var answer = Math.floor(Math.random() * 8) + 1;
    if (answer == 1) {
        alert("It is certain");
    } else if (answer == 2) {
        alert("Outlook good");
    } else if (answer == 3) {
        alert("You may rely on it");
    } else if (answer == 4) {
        alert("Ask again later");
    } else if (answer == 5) {
        alert("Concentrate and ask again");
    } else if (answer == 6) {
        alert("Reply hazy, try again");
    } else if (answer == 7) {
        alert("My reply is no");
    } else if (answer == 8) {
        alert("My sources say no")
    } else {
        alert("That's not a question");}
    alert("The end");
})
```

In JavaScript, the parentheses and curly braces are required because they identify where chunks of code begin and end. The indentations for readability are optional.

The rules are opposite in Python because it doesn't use curly braces or any other special characters to mark the beginning and end of a block of code. The indentations themselves mark those. So those indentations aren't optional — they are required and have a considerable effect on how the code runs. As a result, when you read the code (as a human, not as a computer), it's relatively easy to see what's going on, and you're not distracted by a ton of extra quotation marks. Here is that JavaScript code written in Python:

```
import random
question = input("Ask magic 8 ball a question")
answer = random.randint(1, 8)
if answer == 1:
    print("It is certain")
elif answer == 2:
    print("Outlook good")
elif answer == 3:
```

```
    print("You may rely on it")
elif answer == 4:
    print("Ask again later")
elif answer == 5:
    print("Concentrate and ask again")
elif answer == 6:
    print("Reply hazy, try again")
elif answer == 7:
    print("My reply is no")
elif answer == 8:
    print("My sources say no")
else:
    print("That's not a question")
print("The end")
```

You may have noticed at the top of the Python code the line that starts with `import`. Lines that start with `import` are common in Python, and you'll see why in the next section.

Using Python Modules

One of the secrets to Python's success is that it's comprised of a simple, clean, core language. That's the part you need to learn first. In addition to that core language, many, many modules are available that you can grab for free and access from your own code. These modules are also written in the core language, but you don't need to see that or even know it because you can access all the power of the modules from the basic core language.

Most modules are for some a specific application such as science or artificial intelligence or working with dates and time or . . . whatever. The beauty of using modules is that other people spent a lot of time creating, testing, and fine-tuning that module so you don't have to. You simply import the module into your own Python file, and use the module's capabilities as instructed in the module's documentation.

The preceding sample Magic 8 Ball program starts with this line:

```
import random
```

The core Python language has nothing built into it to generate a random number. Although we could figure out a way to make a random number generator, we don't need to because someone has figured out how to do it and has made the code freely available. Starting your program with `import random` tells the program that you

want to use the capabilities of the random number module to generate a random number. Then, later in the program, you generate a random number between 1 and 8 with this line of code:

```
answer = random.randint(1, 8)
```

Hundreds of free modules for Python are available — you just need to know which ones to import into your program.

Now, you may be wondering where to find all these modules. Well, they're all over the place online. But you'll probably never need to find and download them because you already have the most widely used modules in the world. They were downloaded and installed along with Anaconda. To see for yourself, follow these steps:

1. **Open Anaconda in the usual manner on your computer.**
2. **In the left column, click Environments.**

   On the far right are the Python modules installed on your computer and ready for you to import and use as needed, as shown in Figure 3-4. As you scroll down through the list, you'll see that you already have a ton of them. The rightmost column tells you each module's version.

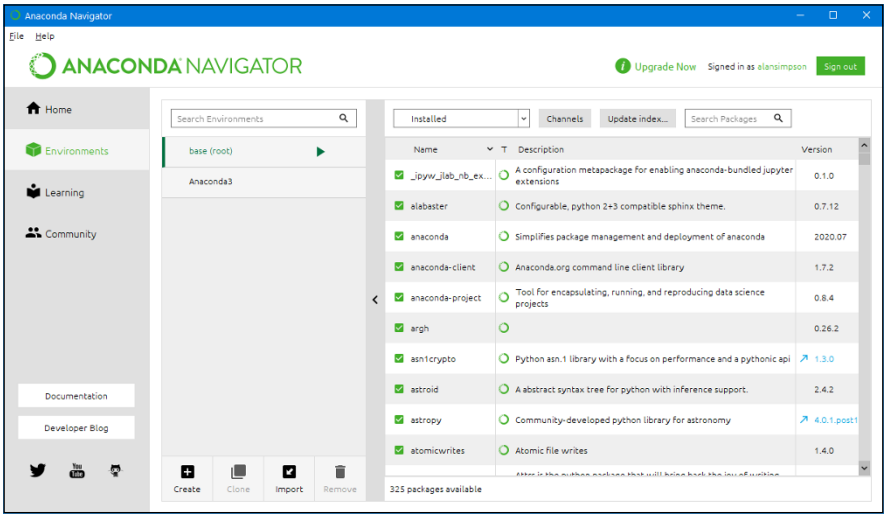


FIGURE 3-4: Installed modules.

You may notice that some version numbers are colored and are preceded with an arrow, which indicates that a more recent version may be available for you to download. As with programming languages, modules evolve over time as their authors improve them and add new capabilities. You're not required to have the

latest version, though. If the version you have is working, you might want to stick with it.

One of many nice things about Anaconda is that to get the latest version, you don't have to do any weird `pip` commands, as many older Python tutorials tell you to do. Instead, just click the arrow or version number of the module or modules you want to download, and then click Apply at the bottom-right corner. Anaconda does all the dirty work of finding the current module, determining whether a newer version is available, and then downloading that version, if it is available.

When all the downloads are finished, you see a dialog box like the one shown in Figure 3-5. If no package names are listed, all the selected modules are up-to-date, so click Cancel and then click Home in the left pane to return to Anaconda's home page. If, on the other hand, package names are listed under The Following Packages Will Be Modified, click Apply to install the latest versions.

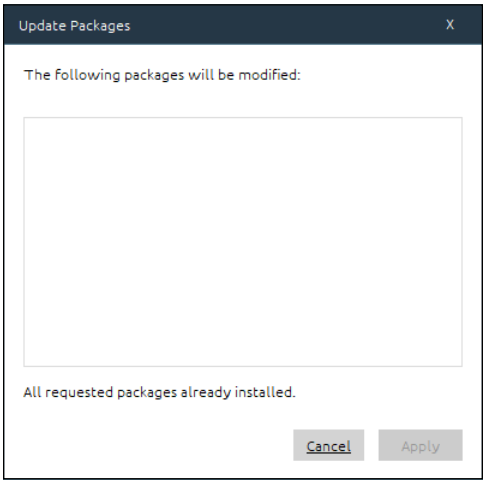


FIGURE 3-5: All our packages are installed and up-to-date.

Understanding the syntax for importing modules

As mentioned, in your own Python code, you must import a module before you can access its capabilities. The syntax for doing so is

```
import modulename [as alias]
```

Code written in a generic format like that, with some parts in italic, some in square brackets, is sometimes called a *syntax chart* because it's not showing you,

literally, what to type. Rather, it's showing the syntax (format) of the code. Here is how information is presented in such a syntax chart:

- The code is case-sensitive, meaning you must type `import` and `as` using all lowercase letters, as shown.
- Anything in italics is a placeholder for information you should supply in your own code. For example, in your code, you would replace *`modulename`* with the name of the module you want to import.
- Anything in square brackets is optional, so you can type the command with or without the part in square brackets.
- You never type the square brackets in your code because they are not part of the Python language. They are used only to indicate optional parts in the syntax.

You can type the `import` line any place you type Python code: at the Python command prompt (`>>>`), in a `.py` file, or in a Jupyter notebook. In a .py file, always put `import` statements first, so their capabilities are available to the rest of the code.

Using an alias with modules

As you just saw with the `import` command's syntax, you can assign an *alias*, or nickname, to any module you import just by following the module name with a space, the word `as`, and a name of your own choosing.

TIP

Most people use a short name that's easy to type and remember, so they don't have to type a long name every time they want to access the module's capabilities.

For example, instead of typing `import random` to import that module, you could import it and give it a nickname such as `rnd`, which is shorter:

```
import random as rnd
```

Then, in subsequent code, you wouldn't use the full name, `random`, to refer to the module. Instead, you'd use the short name, `rnd`:

```
answer = rnd.randint(1, 8)
```

Using an alternative short name may not seem like a big deal in this short example. But some modules have lengthy names, and you might have to refer to the modules in many places in your code.

Now that you've learned some background information, it's time to apply it and start getting your hands dirty with some real Python code. See you in the next chapter.

IN THIS CHAPTER

- Opening the Python file
- Using Python comments
- Understanding data types in Python
- Doing work with Python operators
- Creating variables
- Understanding syntax
- Organizing code

Chapter 4
Building Your First Python Application

So you want to build an application in Python? Whether you want to code a website, analyze data, or create a script to automate something, this chapter gives you the basics you need to get started on your journey. Most people use programming languages like Python to create *application programs,* which are often referred to as *applications* or *apps* or *programs.* To create apps, you need to know how to write code inside a code editor. You also need to start learning the language in which you'll be creating those apps (Python, in this book).

Like any language, you need to understand the individual words so that you can start building sentences and, finally, the blocks of code that will enable your app to work. First, we walk you through creating an app file in which you will create your code. Then you learn the various data types, operators, and variables, which are the words of the Python language, and then Python syntax. Along the way, you see how to save your app, catch mistakes with linting, and comment your code so that you and others can understand how you built it and why.

Are you ready?

Opening the Python App File

You'll be using the ever-popular Visual Studio Code (VS Code) editor in this book to learn Python and create Python apps. We assume that you've already set up your learning and development environment, as described in previous chapters of this minibook, and know how to open the main tools, Anaconda Navigator and VS Code. To follow along in this chapter, start with these steps:

1. **Open Anaconda Navigator and launch VS Code from there.**
2. **If your Python 3 workspace doesn't open automatically, choose File ➪ Open Workspace from the VS Code menu and open the Python 3 workspace you created in Chapter 2.**
3. **Click the `hello.py` file you created in Chapter 2.**
4. **Select all the text on the first line and delete it, so you can start from scratch.**

At this point, `hello.py` should be open in the editor, as shown in Figure 4-1. If any other tabs are open, close them by clicking the X in each.

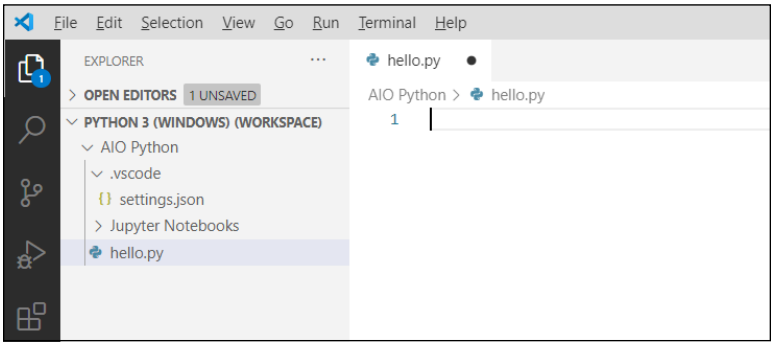


FIGURE 4-1: The `hello.py` file, open for editing in VS Code.

Typing and Using Python Comments

Before you type any code, let's start with a programmer's comment. A *programmer's comment* (usually called a *comment* for short) is text in the program that does nothing. Which brings up the question," If it doesn't *do* anything, why type it in?" As a learner, you can use comments in your code as notes to yourself about what the code is doing. These can help a lot when you're first learning.

However, comments in code aren't strictly for beginners. When working in teams, professionals often use comments to explain to team members what their code is doing. Developers will also put comments in their code as notes to themselves, so that if they review the code in the future, they can refer to their own notes for reminders on why they did something in the code. Because a comment isn't code, your wording can be anything you want. However, to be identified as a comment, you must do one of the following:

- Start the text with a pound sign (#)
- Enclose the text in triple quotation marks

If the comment is short (one line), the leading pound sign is sufficient. Often you'll see the pound sign followed by a space, as in the next example, but the space is optional:

```
# This is a Python comment
```

To type a Python comment into your own code

1. **In VS Code, click next to the 1 under the hello.py tab and type the following:**

   ```
   # This is a Python comment in my first Python app.
   ```

2. **Press Enter.**

   The comment you typed appears on line 1, as shown in Figure 4-2. The comment text will be green if you're using the default color theme. Note that the blinking cursor is now on line 2.

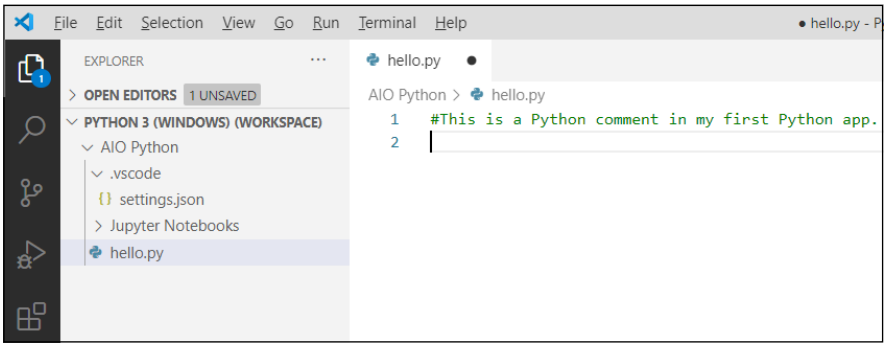


FIGURE 4-2: A comment in `hello.py`.

Although you won't use multiline comments just yet, be aware that you can type longer comments in Python by enclosing them in triple quotation marks. These larger comments are sometimes called *docstrings* and often appear at the top of a Python module, function, class, or method definition, which are app building blocks you will learn about a little later in this book. It isn't necessary to type one right now, but here's an example of what one may look like in Python code:

```
"""This is a multiline comment in Python
This type of comment is sometimes called a docstring.
A docstring starts with three double-quotation marks, and also ends with three
   double quotation marks. """
```

At the beginning and end of the comment, you can use three single quotation marks, rather than three double quotation marks, if you prefer.

In VS Code, comments are usually colored differently than code. Short comments that start with # are green, and docstrings are brown, to help them stand out from the Python code that you run.

You can have an unlimited number of comments in your code. If you're waiting for something to happen after you type a comment . . . don't. When you're working in an editor like this, code doesn't do anything until you run it. And right now, all we have is a comment, so even if we did run this code, nothing would happen because comments are for human readers, not computers. Before you start typing code, you need to start with the absolute basics, which would be . . .

Understanding Python Data Types

You deal with written information all the time and probably don't think about the difference between numbers and text. Numbers are amounts, such as 10 or 123.45. Text consists of letters and words. For computers, the big difference is that they can do arithmetic (add, subtract, multiple, divide) with numbers, but not with letters and words.

For example, everyone knows that 1+1 = 2. The same doesn't apply to letters and words. The expression A+A doesn't necessary equal B or AA or anything else because unlike numbers, letters and words aren't quantities. You can buy *12 apples* at the store, because 12 is a quantity, a number. You can't buy a *snorkel apples* because a *snorkel* is a thing — it's not a quantity, a number, or a scalar value.

Numbers

Numbers in Python must start with a number digit, (0–9); a dot (period), which is a decimal point; or a hyphen (-) used as a negative sign for negative numbers. A number can contain only one decimal point. It should not contain letters, spaces, dollar signs, or anything else that isn't part of a normal number. Table 4-1 shows example of good and bad Python numbers.

TABLE 4-1 Examples of Good and Bad Python Numbers

Number	Good or Bad?	Reason
1	Good	A whole number (integer)
1.1	Good	A number with a decimal point
1234567.89	Good	A large number with a decimal point and no commas
-2	Good	A negative number, as indicated by the starting hyphen
.99	Good	A number that starts with a decimal point because it's less than 1
$1.99	Bad	Contains a $
12,345.67	Bad	Contains a comma
1101 3232	Bad	Contains a space
91740-3384	Bad	Contains a hyphen
123-45-6789	Bad	Contains two hyphens
123 Oak Tree Lane	Bad	Contains spaces and words
(267)555-1234	Bad	Contain parentheses and hyphens
127.0.0.1	Bad	Only one decimal point is allowed

TIP

If you're worried that the number rules won't let you work with dollar amounts, zip codes, addresses, or anything else, stop worrying. You can store and work with *all* kinds of information, as you'll see shortly.

The vast majority of numbers you use will probably match one of the first four examples of good numbers. However, if you happen to be looking at code used for more advanced scientific or mathematical applications, you may occasionally see numbers that contain the letter *e* or the letter *j*. That's because Python supports three different types of numbers, as discussed in the sections that follow.

Integers

An *integer* is any whole number, positive or negative. There is no limit to its size. Numbers such as 0, -1, and 999999999999999 are all perfectly valid integers. From your perspective, an integer is just any valid number that doesn't contain a decimal point.

Floats

A *floating-point number,* often called a *float,* is any valid number that contains a decimal point. Again, there is no size limit: 1.1 and -1.1 and 123456.789012345 are all perfectly valid floats.

If you work with very large scientific numbers, you can put an *e* in a number to indicate the power of 10. For example, 234e1000 is a valid number, and will be treated as a float even if there's no decimal point. If you're familiar with scientific notation, you know 234e3 is 234,000 (replace the *e3* with three zeroes). If you're not familiar with scientific notation, don't worry about it. If you're not using it in your day-to-day work now, chances are you'll never need it in Python either.

Complex numbers

Just about any kind of number can be expressed as an integer or a float, so being familiar with those is sufficient for just about everyone. Note, though, that Python also supports *complex numbers*. These bizarre little charmers always end with the letter *j*, which is the *imaginary* part of the number. If you have no idea what we're talking about, you're normal — only people deep in math land care about complex numbers. If you've never heard of them before now, chances are you won't be using them in your computer work or Python programming.

Words (strings)

Strings are sort of the opposite of numbers. With numbers, you can add, subtract, multiply, and divide because the numbers represent quantities. Strings are for just about everything else. Names, addresses, and all other kinds of text you see every day would be a string in Python (and in computers in general). It's called a *string* because it's a string of characters (letters, spaces, punctuation marks, and maybe some numbers). To us, a string usually has some meaning, such as a person's name or address. But computers don't have eyes to see with or brains to think with or any awareness that humans even exist, so to a computer, if a piece of information is not something on which it can do arithmetic, it's just a string of characters.

Unlike numbers, a string must always be enclosed in quotation marks. You can use either double (") or single (') quotation marks. All the following are valid strings:

```
"Hi there, I am a string"
'Hello world'
"123 Oak Tree Lane"
"(267)555-1234"
"18901-3384"
```

Note that it's fine to use numeric characters (0-9) as well as hyphens and dots (periods) in strings. Each is still a string because it's enclosed in quotation marks.

WARNING

A word of caution. If a string contains an apostrophe (single quote), the entire string should be enclosed in double quotation marks like this:

```
"Mary's dog said Woof"
```

The double quotation marks are necessary because there's no confusion about where the string starts and ends. If you instead used single quotes, like this:

```
'Mary's dog said Woof'
```

the computer would be too dumb to get that right. It would see the first single quote as the start of the string, the next one (after Mary) as the end of the string, and then it wouldn't know what to do with the rest of the stuff and your app wouldn't run correctly.

Similarly, if the string contains double quotation makes, enclose the entire thing in single quotation marks to avoid confusion. For example:

```
'The dog of Mary said "Woof".'
```

The first single quotation mark starts the string, the second one ends it, and the double quotation marks cause no confusion because they're inside the string.

So what if you have a string that contains both single and double quotation marks, like this:

```
Mary's dog said "Woof".
```

This deserves a resounding *hmm*. Fortunately, the creators of Python realized this sort of thing could happen, so they came up with an escape. The solution involves something called *escape characters* because, in a sense, they allow you to escape (avoid) the special meaning of a character such as a single or double quotation

mark. To escape a character, just precede it with a backslash (\). Make sure you use a backslash (the one that leans back toward the previous character, like this \) or it won't work right.

Continuing with the last example, you could enclose the entire thing in single quotation marks, and then escape the apostrophe (which is the same character) by preceding it with a backslash, like this:

```
'Mary\'s dog said "Woof".'
```

Or you could enclose the entire thing in double quotation marks, and escape the quotation marks embedded with the string, like this:

```
"Mary's dog said \"Woof\"."
```

Another common use of the backslash is to use it and *n* (`\n`) to add a line break on the screen where a user is viewing it (the *user* being anyone who uses the app you wrote). For example, this string

```
"The old pond\nA frog jumped in,\nKerplunk!"
```

would look like this when displayed to a user:

```
The old pond
A frog jumped in,
Kerplunk!
```

Each `\n` was converted to a line break.

Booleans

A third data type in Python isn't exactly a number or a string. It's called a *Boolean* (named after a mathematician named George Boole), and it can be one of two values: either `True` or `False`. It may seem odd to have a data type for something that can only be `True` or `False`, but doing so is efficient because you can store the `True` or `False` value using a single bit, which is the smallest unit of storage in a computer.

In Python code, people store `True` and `False` values in *variables* (placeholders in code that we discuss later in this chapter) using a format similar to this:

```
x = True
```

Or perhaps this:

```
x = False
```

You know `True` and `False` are Boolean here because they are not enclosed in quotation marks (as a string would be) and are not numbers. Also, the *initial cap* is required. In other words, the Boolean values `True` and `False` must be written as shown.

Working with Python Operators

As we discuss in the preceding section, with Python and computers in general, it helps to think of information as being one of the following data types: number, string, or Boolean. You also use computers to *operate* on that information, meaning do any necessary math or comparisons or searches or whatever to help you find information and organize it in a way that makes sense to you.

Python offers many different *operators* for working with and comparing types of information. Here we summarize them all for future reference, without going into great detail. Whether you use an operator in your own work depends on the types of apps you develop. For now, it's sufficient just to be aware that they're available.

Arithmetic operators

Arithmetic operators, as the name implies, are for doing arithmetic; addition, subtraction, multiplication, division, and more. Table 4-2 lists Python's arithmetic operators.

TABLE 4-2 Python's Arithmetic Operators

Operator	Description	Example
+	Addition	1 + 1 = 2
-	Subtraction	10 - 1 = 9
*	Multiplication	3 * 5 = 15
/	Division	10 / 5 = 2
%	Modulus (remainder after division)	11 % 5 = 1
**	Exponent	3**2 = 9
//	Floor division	11 // 5 = 2

The first four items in the table are the same as you learned in elementary school. The last three are a little more advanced, so we'll explain them here:

- The *modulus* is the remainder after division. So, for example, 11 % 5 is 1 because if you divide 11 by 2 you get 5 remainder 1. That 1 is the modulus (sometimes called the *modulo*).
- The *exponent* is ** because you can't type a small raised number in code. But it just means "raised to the power of." For example, 3**2 is 3^2 (or 3 squared), which is 3*3, or 9, and 3**4 is 3*3*3*3, or 81.
- *Floor division*, indicated by //, is integer division in that anything after the decimal point is *truncated* (cut off), without any rounding. For example, in regular division 9/5 is 1.8. But 9//5 is 1 because the .8 is just chopped off — it isn't rounded to 2.

Comparison operators

Computers can make decisions as part of doing their work. But these decisions are not judgement call decisions or anything human like that. These decisions are based on absolute facts that are based on comparisons. The *comparison operators* Python offers to help you write code that makes decisions are listed in Table 4-3.

TABLE 4-3 Python Comparison Operators

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
is	Object identity
is not	Negated object identity

The first few are self-explanatory, so we won't go into detail there. The last two are tricky because they concern Python objects, which we haven't talked about yet. Talking about Python objects right now would be a big digression, so if you're at all confused about any operators right now, don't worry about it.

Boolean operators

The *Boolean operators* work with Boolean values (`True` or `False`) and are used to determine if one or more things is `True` or `False`. Table 4-4 summarizes the Boolean operators.

TABLE 4-4 Python Boolean Operators

Operator	Code Example	What It Determines
or	x or y	Either x or y is True
and	x and y	Both x and y are True
not	not x	x is not True

Python Style Guide (PEP 8) recommends always putting whitespace around operators. In other words, you want to use the spacebar on the keyboard to put a space before the operator, type the operator, and then add another space before continuing the line of code. Here is a somewhat simple example. We know you're not familiar with coding just yet so don't worry too much about the meaning of the code. Instead, note the spaces around the = and > (greater than) operators:

```
num = 10
if num > 0:
    print("Positive number")
else:
    print("Negative number")
```

The first line stores the number `10` in a variable named `num`. Then the `if` checks to see whether `num` is greater than (`>`) `0`. If it is, the program prints `Positive number`. Otherwise, it prints `Negative number`. So, let's say you change the first line of the program to this:

```
num = -1
```

If you make that change and run the program again, it prints `Negative number` because `-1` is a negative number.

We used `num` as a sample variable name in this example so we could show you some operators with space around them. Of course, we haven't told you what variables are, so that part of the example may have left you scratching your head. We clear up that part of this business next.

Creating and Using Variables

Variables are a big part of Python and all computer programming languages. A *variable* is simply a placeholder for information that may vary (change). For example, when you go to Amazon's home page, you can see your name and the date you became a customer, as shown in Figure 4-3. The screen may look different when you visit, but the basic information should be on the page somewhere. Both those pieces of information are variables, because they change depending on who is signed in to Amazon.

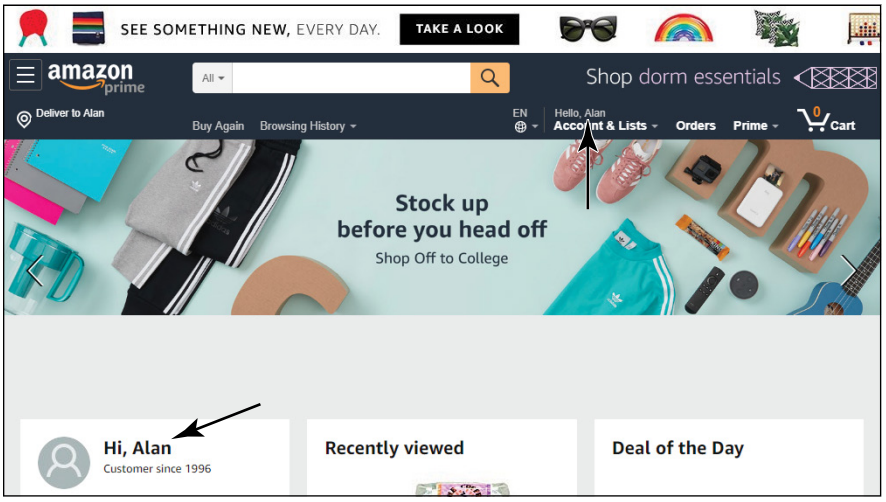


FIGURE 4-3: Your name and the date you became a customer appear on Amazon's home page.

Certainly not everyone who goes to Amazon that day is named Alan and has been a member since 1996. Other people must be seeing other stuff there. But Amazon certainly can't make a custom home page for every one of its millions of users. Most of what's on that page is probably *literal* — meaning everyone who views the page sees the same stuff. Only the information that changes depending on who is viewing the page is stored as a variable.

In your code, a variable is represented by a variable *name* rather than a specific piece of information. Here is another way to think of it. Anytime you buy one or more of some product, the extended price is the unit price times the number of items you bought. In other words

*Quantity * Unit Price = Extended Price*

You can consider Quantity and Unit Price to be variables because no matter what numbers you plug in for Quantity and Unit Price, you get the correct extended

price. For example, if you buy three turtle doves for $1.00 apiece, your extended price is $3.00 (3 * $1.00). If you buy two dozen roses for $1.50 apiece, the extended price is $36 because 1.5 * 24 is 36.

Creating valid variable names

In our explanation of variables, we used names like Quantity and Unit Price, and this is fine for a general example. In Python, you can also make up your own variable names, but they must conform to the following rules to be recognized as variable names:

- The variable name must start with a letter or an underscore (_).
- After the first character, you can use letters, numbers, or underscores.
- Variable names are case sensitive, so after you make up a name, any reference to that variable must use the same uppercase and lowercase letters.
- Variable names cannot be enclosed in, or contain, single or double quotation marks.
- PEP 8 style conventions recommend that you use only lowercase letters in variable names and use an underscore to separate multiple words.

PEP 8, which we mentioned in previous chapters, is a style guide for writing code, rather than strict must-follow rules. So you often see variable names that don't conform to that last style. *Camel case* formatting — whereby the first letter is lowercase and new words are capitalized — is common, even in Python, for example, `extendedPrice` or `unitPrice`.

Experienced Python purists sometimes get a disgusted look on their face when they see names like these in your code. They would prefer you stick with the PEP 8 style guidelines, which recommend using `extended_price` and `unit_price` as your variable names, on the grounds that the PEP 8 syntax is more readable for human programmers.

Creating variables in code

To create a variable, you use the following syntax (order of things):

```
variablename = value
```

where *`variablename`* is the name you make up. You can use *x* or *y*, as people often do in math, but in larger programs, it's a good idea to give your variables more meaningful names, such as `quantity` or `unit_price` or `sales_tax` or `user_name`, so that you can remember what you're storing in the variable.

The *value* is whatever you want to store in the variable. It can be a number, a string, or a Boolean `True` or `False` value.

The = sign is the *assignment operator* and is so named because it assigns the value (on the right) to the variable (on the left). For example, in the following:

```
x = 10
```

we are storing the number `10` in a variable named `x`. In other words, we're assigning the value `10` to the `x` variable.

And here:

```
user_name = "Alan"
```

we're putting the string `Alan` in a variable named `username`.

Manipulating variables

Much of computer programming revolves around storing values in variables and manipulating that information with operators. Time to try some simple examples to get the hang of it. If you still have VS Code open with that one comment displayed, follow these steps in the VS Code editor:

1. **Under the line that reads** *`# This is a Python comment in my first Python app.`***, type this comment and press Enter:**

   ```
   # This variable contains an integer
   ```

2. **Type the following (don't forget to put a space before and after the = sign) and press Enter:**

   ```
   quantity = 10
   ```

3. **Type the following and press Enter:**

   ```
   # This variable contains a float
   ```

4. **Type the following (don't type a dollar sign!) and press Enter:**

   ```
   unit_price = 1.99
   ```

5. **Type the following and press Enter:**

```
# This variable contains the result of multiplying quantity times unit price
```

6. **Type the following (with spaces around the operators) and press Enter:**

```
extended_price = quantity * unit_price
```

7. **Type the following and press Enter:**

```
# Show the results
```

8. **Finally, type this and press Enter:**

```
print(extended_price)
```

Your Python app creates some variables, stores some values in them, and calculates a new value, `extended_price`, based on the contents of the `quantity` and `unit_price` variables. The last line displays the contents of the `extended_price` variable on the screen. Remember, the comments don't *do* anything in the program as it's running. The comments are just notes to yourself about what's going on in the program.

Figure 4-4 shows how things should look now. If you made any errors, you may see some wavy lines near errors or stylistic suggestions, such as an extra space or an omitted Enter at the end of a line. When typing code, you must be accurate. You can't type something that looks sort of like what you were supposed to type. When texting to humans, you can make all kinds of typographical errors and your human recipient can usually figure out what you meant based on the context of the message. But computers don't have eyes or brains or a concept of context, so they will generally just not work properly if your code has errors.

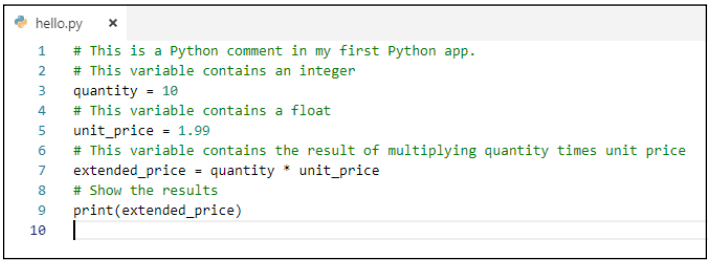


FIGURE 4-4: Your first Python app typed into VS Code.

In other words, if the code is wrong, it won't work when you run it. It's as simple as that — no exceptions.

Saving your work

Typing code is like typing other documents on a computer. If you don't save your work, you may not have it the next time you sit down at your computer and go looking for it. So if you haven't enabled Auto Save on the File menu, as discussed in Chapter 2 of this minibook, choose File➪Save.

Running your Python app in VS Code

Now you can run the app and see if it works. An easy way to do that is to right-click the `hello.py` filename in the Explorer bar and choose Run Python File in Terminal, as shown in Figure 4-5.

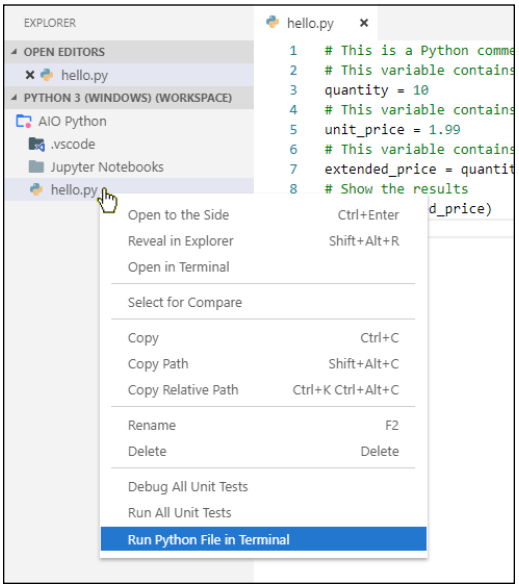


FIGURE 4-5: Right-click a `.py` file and choose Run Python File in Terminal.

If your code is typed correctly, you should see the result, `19.9`, in the Terminal window, as shown in Figure 4-6. The result is the output from `print(extended_price)` in the code, and it's `19.9` because the quantity (`10`) times the unit price (`1.99`) is `19.9`.

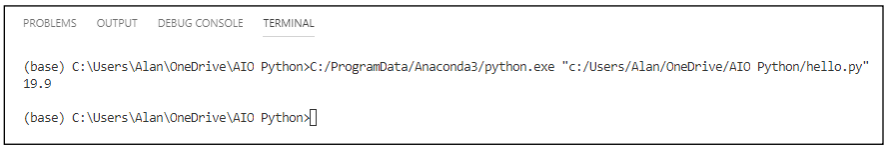


FIGURE 4-6: The 19.9 is the output from `print(extended_price)` in the code.

Suppose your app must calculate the total cost of 14 items that each cost $26.99. Can you think of how to make that happen? You certainly wouldn't need to write a whole new app. Instead, in the code you're working with now, change the value of the quantity variable from `10` to `14`. Change the value of the `unitprice` variable to 26.99 (remember, no dollar signs in your number). Here's how the code looks with those changes:

```
# This is a Python comment in my first Python app.
# This variable contains an integer
quantity = 14
# This variable contains a float
unit_price = 26.99
# This variable contains the result of multiplying quantity times unit price
extended_price = quantity * unit_price
# Show some results on the screen.
print(extended_price)
```

Save your work (unless you've turned on AutoSave). Then run the app by right-clicking and choosing Run Python File in Terminal once again — just like the first time. The results are again quite a bit of gobbledygook. But you should see the correct answer, 377.85999999999996, in the Terminal window near the bottom of the VS Code window. It doesn't round to pennies and it doesn't even look like a dollar amount. But you need to learn to crawl before you can learn to pole vault, so for now just be happy with getting your apps to run.

Understanding What Syntax Is and Why It Matters

If you look up *syntax* in the dictionary, one definition you might find is "the arrangement of words and phrases to create well-formed sentences in a language." In programming languages like Python, there is no such thing as a well-formed sentence. But Python does have words in the sense that you need a space between each word, just as you do when typing regular text like this, and the order of those words is important.

Syntax is important in human languages because order contributes much to the meaning. For example, compare these three short sentences:

Mary kissed John.

John kissed Mary.

Kissed Mary John.

All three sentences contain the same words, but the meanings are different. The first two make it clear who kissed whom, and the last one is a little hard to interpret.

Proper syntax in programming languages is every bit as important as it is in human languages — even more so, in some ways, because when you make a mistake speaking or writing to someone, that other person can usually figure out what you meant by the context of your words. But computers aren't nearly that smart. Computers don't have brains, can't guess your actual meaning based on context, and in fact the concept of context doesn't even exist for computers. So syntax matters even more in programming languages than in human languages.

Looking back at the earliest code in this chapter, note that all the lines of actual code (not the comments, which start with #) follow this syntax:

```
variablename = value
```

where *variablename* is some name you made up, and *value* is something you are storing in that variable. It works because it's the proper syntax. If you try to do it like this, it won't work:

```
value = variablename
```

For example, the following is the correct way to store the value `10` in a variable named `x`:

```
x = 10
```

It might seem you could also do it the following way, but it won't work in Python:

```
10 = x
```

If you run the app with that line in it, nothing terrible will happen — you won't break anything. But you will get an error message like the following:

```
File ".../AIO Python/hello.py", line 10
10=x
^
SyntaxError: cannot assign to literal
```

The `SyntaxError` part tells you that Python doesn't know what to do with that line of code because you didn't follow the proper syntax. To fix the error, just rewrite the line as

```
x = 10
```

Now let's talk about individual lines of code. In Python, a line of code ends with a line break or a semicolon. For example, this is three lines of Python code:

```
first_name = "Alan"
last_name = "Simpson"
print(first_name, last_name)
```

It would also be acceptable to use a semicolon instead of a line break:

```
first_name = "Alan"; last_name = "Simpson"
print(first_name, last_name)
```

Or, if you prefer:

```
first_name = "Alan"; last_name = "Simpson"; print(first_name, last_name)
```

The code runs the same whether you end each line with a break or a semicolon.

Note how the variable names are all lowercase, and the words are separated by an underscore:

```
first_name
last_name
```

Using all lowercase letters for variable names with words separated by underscores is a *naming convention* in Python. But note that a *convention* is not the same as a *syntax rule*. You could name the variables as follows without breaking any syntax rules:

```
FirstName
LastName
```

The naming convention tries to get programmers to follow basic stylistic guidelines that make the code more readable to other programs, which is especially important when working in programming teams or groups.

So far you've looked at lines of code. There are also *code blocks* where two or more lines of code work together. Here is an example:

```
x = 10
if x == 0:
    print("x is zero")
else:
    print("x is ",x)
print("All done")
```

The == (two equal signs) means "is equal to" in Python and is used to compare values to one another to see if they're equal. That's different from just = (one equal sign), which is the assignment operator for assigning variables.

The first line, `x = 10`, is just a line of code. Next, the `if x == 0` tests to see whether the `x` variable contains the number `0`. If `x` *does* contain `0`, the indented line (`print("x is zero")`) executes and that's what you see on the screen. However, if `x` does not contain `0`, that indented line is skipped and the `else:` statement executes. The indented line under else: `print("x is ",x)` executes, but *only* if the `x` doesn't contain `0`. The last line, `print("All done!")`, executes no matter what, because it's not indented.

So, as you can see, indentations matter a lot in Python. In the preceding code, only one of the indented lines will execute depending on the value in `x`. You learn about the specifics of using indentations in your code as you progress through the book. For now, just try to remember that syntax and indentations are important in Python, so you must type carefully when writing code.

If you have linting and PEP 8 enabled in your Workspace settings, as described in Chapter 3, you may see wavy underlines in code that appears to be okay. Hovering the mouse pointer over such an underline will usually show a message indicating the problem, as shown in Figure 4-7.

The exact wording and syntax of any error might vary, depending on the version of linting you're using. But as an example, in Figure 4-7, the first part of the message, `[pep8]`, tells you that this error is related to PEP 8 syntax, which says you should put whitespace around operators:

```
[pep8] missing whitespace around operator
```

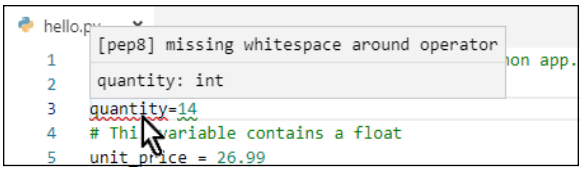


FIGURE 4-7: Touching the mouse pointer to a red wavy underline.

The second part just tells you that the variable named `quantity` contains an integer (`int`), which is a whole number. That part of the message is information, not an error.

To fix the error, put whitespace around the = sign. In other words, use the spacebar on your keyboard to put a space before and after the = sign.

But now you see a wavy underline under the `14`. What's up with that? Well, to find out, simply click or hover the mouse pointer over the green wavy underline and leave the mouse pointer sitting right there until you see an explanation, as in Figure 4-8.

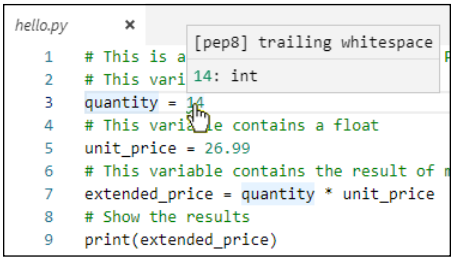


FIGURE 4-8: Touching the mouse pointer to a green wavy underline.

Again, the exact wording of the message may change by the time you read this. But in this example, the message is `[pep8] trailing whitespace` on top and `14: int` on the bottom. The bottom part is just information, telling you that `14` is stored as an integer. The error is the trailing whitespace. In other words, there's a space after the `14` on that line. You can't see it, because it's just a space. To eliminate trailing spaces and fix the error, click the end of that line and press Backspace until the cursor is right up to the `4` in `14`.

Other colored errors are stylistic errors. But you won't know the specific error until you hover the mouse pointer over the wavy underline and leave the mouse pointer there until you see the message. And the error won't go away until you take whatever action is required to fix it.

TIP

If PEP 8 errors seem overwhelming while you're trying to learn, turn them off temporarily. Choose File➪Settings (Windows) or Code➪Settings (Mac). Then in code view, set python.linting.pyLintEnabled or python.linting.pcodestyleEnabled or both to false.

Putting Code Together

The exercises you've just completed explain how to type, save, run, and change an app, save it again, and run it again. Those tasks define what you'll be doing with any kind of software development in any language, so you should practice them until they become second nature. But don't worry: You don't have to do this one chapter over and over again to get the hang of it. You'll be using these same skills throughout this book as you work your way from beginner to hot-shot twenty-first-century Python developer.

2 Understanding Python Building Blocks

Contents at a Glance

IN THIS CHAPTER

- Mastering whole numbers
- Juggling numbers with decimal points
- Simplifying strings
- Conquering Boolean True/False
- Working with dates and times

Chapter 1

Working with Numbers, Text, and Dates

Computer languages in general, and certainly Python, deal with information in ways that are different from what you may be used to in your everyday life. This idea takes some getting used to. In the computer world, *numbers* are numbers you can add, subtract, multiply, and divide. Python also differentiates between whole numbers (integers) and numbers that contain a decimal point (floats). Words (textual information such as names and addresses) are stored as strings, which is short for "a string of characters." In addition to numbers and strings, there are Boolean values, which can be either `True` or `False`.

In real life, we also have to deal with dates and times, which are yet another type of information. Python doesn't have a built-in data type for dates and times, but thankfully, a free module you can import any time works with such information. This chapter is all about taking full advantage of the various Python data types.

Calculating Numbers with Functions

A *function* in Python is similar to a function on a calculator, in that you pass something into the function, and the function passes something back. For example, most calculators and programming languages have a square root function: You give them a number, and they give back the square root of that number.

Python functions generally have the syntax:

```
variablename = functionname(param[,param])
```

Because most functions return some value, you typically start by defining a variable to store what the function returns. Follow that with the = sign and the function name, followed by a pair of parentheses. Inside the parentheses you may pass one or more values (called *parameters*) to the function.

For example, the `abs()` function accepts one number and returns the absolute value of that number. If you're not a math nerd, this just means if you pass it a negative number, it returns that same number as a positive number. If you pass it a positive number, it returns the same number you passed it. In other words, the `abs()` function simply converts negative numbers to positive numbers.

As an example, in Figure 1-1 (which you can try out for yourself hands-on in a Jupyter notebook, at the Python prompt, or in a `.py` file in VS Code), we created a variable named `x` and assigned it the value `-4`. Then we created a variable named `y` and assigned it the absolute value of `x` using the `abs()` function. Printing `x` shows its value, `-4`, which hasn't changed. Printing `y` shows `4`, the absolute value of `x` as returned by the `abs()` function.

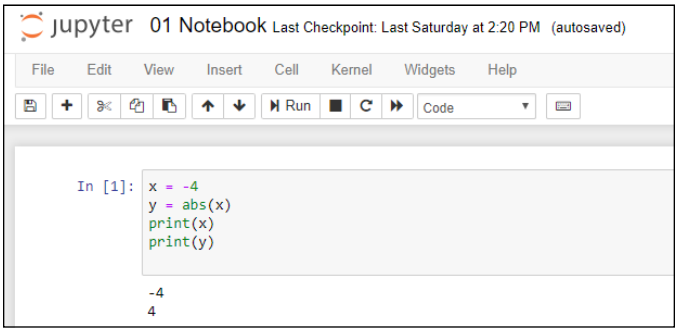


FIGURE 1-1: Trying out the `abs()` function.

Even though a function always returns one value, some functions accept two or more values. For example, the `round()` function takes one number as its first argument. The second argument is the number of decimal places to which you want to round that number, for example, `2` for two decimal places. In the example in Figure 1-2, we created a variable, `x`, with a whole lot of digits after the decimal point. Then we created a variable named `y` to return the same number rounded to two decimal places. Then we printed both results.

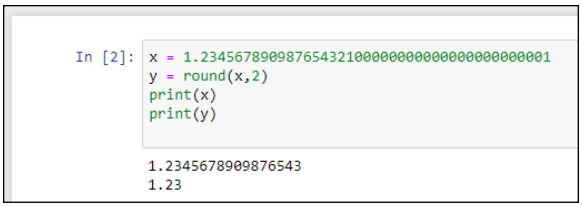


FIGURE 1-2: Trying out the `round()` function.

Python has many built-in functions for working with numbers, as shown in Table 1-1. Some may not mean much to you if you're not into math in a big way, but don't let that intimidate you. If you don't understand what a function does, chances are it's not doing something relevant to the kind of work you do. But if you're curious, you can always search the web for *python* followed by the function name for more information. For a more extensive list, search for *python 3 built-in functions*.

TABLE 1-1 Some Built-In Python Functions for Numbers

Built-In Function	Purpose
`abs(x)`	Returns the absolute value of number *x* (converts negative numbers to positive).
`bin(x)`	Returns a string representing the value of *x* converted to binary.
`float(x)`	Converts a string or number *x* to the float data type.
`format(x, y)`	Returns *x* formatted according to the a pattern specified in *y*. This older syntax has been replaced with f-strings in current Python versions.
`hex(x)`	Returns a string containing *x* converted to hexadecimal, prefixed with 0x.
`int(x)`	Converts *x* to the integer data type by truncating (not rounding) the decimal portion and any digits after it.
`max(x, y, z, ...)`	Takes any number of numeric arguments and returns whichever is the largest.
`min(x, y, z, ...)`	Takes any number of numeric arguments and returns whichever is the smallest.
`oct(x)`	Converts *x* to an octal number, prefixed with 0o to indicate octal.
`round(x, y)`	Rounds the number *x* to *y* number of decimal places.
`str(x)`	Converts the number *x* to the string data type.
`type(x)`	Returns a string indicating the data type of *x*.

Figure 1-3 shows examples of proper Python syntax for using the built-in math functions.

```
pi=3.14159265358979
x=128
y=-345.67890987
z=-999.9999
print(abs(z))
print(int(z))
print(int(abs(z)))
print(round(pi,4))
print(bin(x))
print(hex(x))
print(oct(x))
print(max(pi,x,y,z))
print(min(pi,x,y,z))
print(type(pi))
print(type(x))
print(type(str(y)))
```

```
999.9999
-999
999
3.1416
0b10000000
0x80
0o200
128
-999.9999
<class 'float'>
<class 'int'>
<class 'str'>
```

FIGURE 1-3: Playing around with built-in math functions at the Python prompt.

You can also *nest* functions — meaning you can put functions inside functions. For example, when `z = -999.9999`, the expression `print(int(abs(z)))` prints the integer portion of the absolute value of `z`, which is `999`. The original number is converted to positive, and then the decimal point and everything to its right chopped off.

Still More Math Functions

In addition to the built-in functions you've learned about so far, still others you can import from the `math` module. If you need them in an app, put `import math` near the top of the `.py` file or Jupyter cell to make those functions available to the rest of the code. Or to use them at the command prompt, first enter the **`import math`** command.

One of the functions in the `math` module is the `sqrt()` function, which gets the square root of a number. Because it's part of the `math` module, you can't use it without importing the module first. For example, if you enter the following, you'll get an error because `sqrt()` isn't a built-in function:

```
print(sqrt(81))
```

Even if you do two commands like the following, you'll still get an error because you're treating `sqrt()` as a built-in function:

```
import math
print(sqrt(81))
```

To use a function from a module, you have to import the module *and* precede the function name with the module name and a dot. So let's say you have some value, `x`, and you want the square root. You have to import the `math` module and use `math.sqrt(x)` to get the correct answer, as shown in Figure 1-4. Entering that command shows 9.0 as the result, which is indeed the square root of 81.

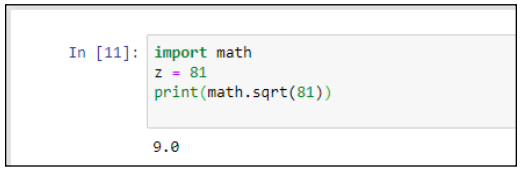


FIGURE 1-4: Using the `sqrt()` function from the `math` module.

The `math` module offers a lot of trigonometric and hyperbolic functions, powers and logarithms, angular conversions, constants such as `pi` and `e`. We won't delve into all of them because advanced math isn't relevant to most people. You can check them all out anytime by searching the web for *python 3 math module functions*. Table 1-2 offers examples that may prove useful in your own work.

TABLE 1-2 Some Functions from the Python Math Module

Built-In Function	Purpose
`math.acos(x)`	Returns the arccosine of `x` in radians
`math.atan(x)`	Returns the arctangent of `x`, in radians
`math.atan2(y, x)`	Converts rectangular coordinates (x, y) to polar coordinates (r, theta)
`math.ceil(x)`	Returns the ceiling of `x`, the smallest integer greater than or equal to `x`
`math.cos(x)`	Returns the cosine of `x` radians
`math.degrees(x)`	Converts angle `x` from radians to degrees
`math.e`	Returns the mathematical constant *e* (2.718281 . . .)
`math.exp(x)`	Returns *e* raised to the power *x*, where *e* is the base of natural logarithms

(continued)

TABLE 1-2 *(continued)*

Built-In Function	Purpose
`math.factorial(x)`	Returns the factorial of `x`
`math.floor()`	Returns the floor of `x`, the largest integer less than or equal to `x`
`math.isnan(x)`	Returns `True` if `x` is not a number; otherwise returns `False`
`math.log(x, y)`	Returns the logarithm of `x` to base `y`
`math.log2(x)`	Returns the base-2 logarithm of `x`
`math.pi`	Returns the mathematical constant pi (3.141592 . . .)
`math.pow(x, y)`	Returns `x` raised to the power `y`
`math.radians(x)`	Converts angle `x` from degrees to radians
`math.sin(x)`	Returns the sine of `x`, in radians
`math.sqrt(x)`	Returns the square root of `x`
`math.tan(x)`	Returns the tangent of `x` radians
`math.tau()`	Returns the mathematical constant tau (6.283185 . . .)

The constants `pi`, `e`, and `tau` are unusual for functions in that you don't use parentheses. As with any function, you can use these functions in expressions (calculations) or assign their values to variables. Figure 1-5 shows some examples of using functions from the `math` module.

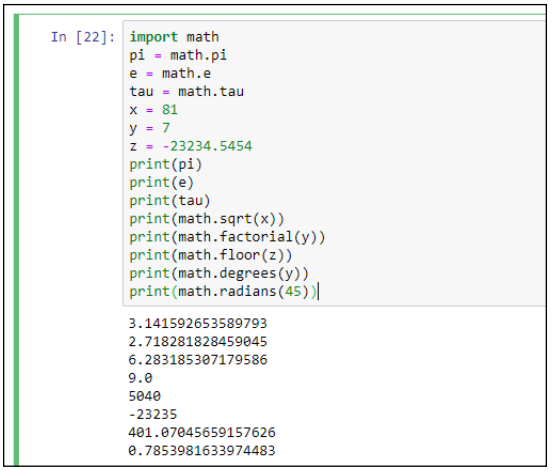


FIGURE 1-5: More playing around with built-in math functions at the Python prompt.

Formatting Numbers

Over the years, Python has offered different methods for displaying numbers in formats familiar to us humans. For example, most people would rather see dollar amounts expressed in the format $1,234.56 rather than 1234.56006595069540569540595959. The easiest way to format numbers in Python, starting with version 3.6, is to use f-stings.

Formatting with f-strings

Format strings, or f-strings, are the easiest way to format data in Python. All you need is a lowercase *f* or uppercase *F* followed immediately by some text or expressions enclosed in quotation marks. Here is an example:

```
f"Hello {username}"
```

The `f` before the first quotation mark tells Python that what follows is a format string. Inside the quotation marks, the text, called the *literal part*, is displayed literally (exactly as typed in the f-string). Anything in curly braces is the *expression part* of the f-string, a placeholder for what will appear when the code executes. Inside the curly braces, you can have an *expression* (a formula to perform some calculation, a variable name, or a combination of the two). Here is an example:

```
username = "Alan"
print(f"Hello {username}")
```

When you run this code, the `print` function displays the word `Hello`, followed by a space, followed by the contents of the `username` variable, as shown in Figure 1-6.

FIGURE 1-6: A super simple f-string for formatting.

Here is another example of an expression — the formula `quantity` times `unit_price` — inside the curly braces:

```
unit_price = 49.99
quantity = 30
print(f"Subtotal: ${quantity * unit_price}")
```

The output from that, when executed, follows:

```
Subtotal: $1499.7
```

That `$1499.7` isn't an ideal way to show dollar amounts. Typically, we like to use commas in the thousands places, and two digits for the pennies, as in the following:

```
Subtotal: $1,499.70
```

Fortunately, f-strings provide you with the means to do this formatting, as you learn next.

Showing dollar amounts

To get a comma to appear in the dollar amount and the pennies as two digits, you can use a *format string* inside the curly braces of an expression in an f-string. The format string starts with a colon and needs to be placed inside the closing curly brace, right up against the variable name or the value shown.

To show commas in thousands places, use a comma in your format string right after the colon, like this:

```
:,
```

Using the current example, you would do the following:

```
print(f"Subtotal: ${quantity * unit_price:,}")
```

Executing this statement produces this output:

```
Subtotal: $1,499.7
```

To get the pennies to show as two digits, follow the comma with

```
.2f
```

The `.2f` means "two decimal places, fixed" (never any more or less than two decimal places). The following code will display the number with commas and two decimal places:

```
print(f"Subtotal: ${quantity * unit_price:,.2f}")
```

Here's what the code displays when executed:

```
Subtotal: $1,499.70
```

Perfect! That's exactly the format we want. So anytime you want to show a number with commas in the thousands places and exactly two digits after the decimal point, use an f-string with the format string, `.2f`.

Formatting percent numbers

Now, suppose your app applies sales tax. The app needs to know the sales tax rate, which should be expressed as a decimal number. So if the sales tax rate is 6.5 percent, it has to be written as 0.065 (or .065, if you prefer) in your code, like this:

```
sales_tax_rate = 0.065
```

It's the same amount with or without the leading zero, so just use whichever format works for you.

This number format is ideal for Python, and you wouldn't want to mess with that. But if you want to display that number to a human, simply using a `print()` function displays it exactly as Python stores it:

```
sales_tax_rate = 0.065
print(f"Sales Tax Rate {sales_tax_rate}")
Sales Tax Rate 0.065
```

When displaying the sales tax rate for people to read, you'll probably want to use the more familiar 6.5% format rather than .065. You can use the same idea as with fixed numbers (`.2f`). However, you replace the `f` for fixed numbers with `%`, like this:

```
print(f"Sales Tax Rate {sales_tax_rate:.2%}")
```

Running this code multiples the sales tax rate by 100 and follows it with a % sign, as you can see in Figure 1-7.

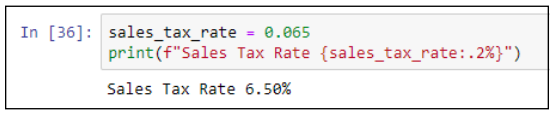


FIGURE 1-7: Formatting a percentage number.

In both of the previous examples, we used `2` for the number of digits. But of course you can display any number of digits you want, from zero (none) to whatever level of precision you need. For example, using `.1%`, as in the following:

```
print(f"Sales Tax Rate {sales_tax_rate:.1%}")
```

displays this output when the line is executed:

```
Sales Tax Rate 6.5%
```

Replacing `1` with a `9`, like this:

```
print(f"Sales Tax Rate {sales_tax_rate:.9%}")
```

displays the percentage with nine digits after the decimal point:

```
Sales Tax Rate 6.500000000%
```

You don't need to use an f-string only inside a call to the `print` function. You can also execute an f-string and save the result in a variable that you can display later. The format string itself is like any other string in that it must be enclosed in single, double, or triple quotation marks. When using triple quotation marks, you can use either three single quotation marks or three double quotation marks. It doesn't matter which you use as the outermost quotation marks on the format string; the output is the same, as you can see in Figure 1-8.

TIP

For single and double quotation marks, use the corresponding keyboard keys. For triple quotation marks, you can use three of either. Make sure you end the string with exactly the same characters you used to start the string. For example, all the strings in Figure 1-8 are perfectly valid code, and they will all be treated the same.

```
sales_tax_rate = 0.065
sample1 = f'Sales Tax Rate {sales_tax_rate:.2%}'
sample2 = f"Sales Tax Rate {sales_tax_rate:.2%}"
sample3 = f"""Sales Tax Rate {sales_tax_rate:.2%}"""
sample4 = f'''Sales Tax Rate {sales_tax_rate:.2%}'''

print(sample1)
print(sample2)
print(sample3)
print(sample4)
```

```
Sales Tax Rate 6.50%
Sales Tax Rate 6.50%
Sales Tax Rate 6.50%
Sales Tax Rate 6.50%
```

FIGURE 1-8: An f-string can be encased in single, double, or triple quotation marks.

Making multiline format strings

If you want to have multiline output, you can add line breaks to your format strings in a few ways:

- **Use `/n`:** You can use a single-line format string with `\n` any place you want a line break. Just make sure you put the `\n` in the literal portion of the format string, not inside curly braces. For example:

```
user1 = "Alberto"
user2 = "Babs"
user3 = "Carlos"
output=f"{user1} \n{user2} \n{user3}"
print(output)
```

  When executed, this code displays:

```
Alberto
Babs
Carlos
```

- **Use triple quotation marks (single or double):** If you use triple quotation marks around your format string, you don't need to use `\n`. You can just break the line in the format string wherever you want it to break in the output. For example, look at the code in Figure 1-9. The format string is in triple quotation marks and contains multiple line breaks. The output from running the code has line breaks in the same places.

```
unit_price = 49.95
quantity = 32
sales_tax_rate = 0.065
subtotal = quantity * unit_price
sales_tax = sales_tax_rate * subtotal
total = subtotal + sales_tax
output=f"""
Subtotal:  ${subtotal:,.2f}
Sales Tax: ${sales_tax:,.2f}
Total:     ${total:,.2f}
"""
print(output)
```

```
Subtotal:  $1,598.40
Sales Tax: $103.90
Total:     $1,702.30
```

FIGURE 1-9: A multiline f-string enclosed in triple quotation marks.

As you can see, the output honors the line breaks and even the blank spaces in the format string. Unfortunately, it's not perfect — in real life, we would right align the numbers so that the decimal points line up. All is not lost, though, because with format strings you can also control the width and alignments of your output.

Formatting width and alignment

You can also control the width of your output (and the alignment of content within that width) by following the colon in your f-string with < (for left aligned), ^ (for centered), or > (for right aligned). Put any of these characters right after the colon in your format string. For example, the following will make the output 20 characters wide, with the content right aligned:

```
:>20
```

In Figure 1-9, all the dollar amounts are left aligned, because that's the default. To right align numbers, which is how we usually see dollar amounts, you can use > in an f-string. To make the numbers the same width, specify a number after the > character. For example, in Figure 1-10, each f-string includes >9, which causes each displayed number to be right aligned and 9 characters wide. The output, which you can see at the bottom of the figure, makes all the numbers align to the right, with their dollar signs neatly aligned to the left. The spaces to the right of each dollar sign make sure each number is exactly 9 characters wide.

```
unit_price = 49.95
quantity = 32
sales_tax_rate = 0.065
subtotal = quantity * unit_price
sales_tax = sales_tax_rate * subtotal
total = subtotal + sales_tax
output=f"""
Subtotal:  ${subtotal:>9,.2f}
Sales Tax: ${sales_tax:>9,.2f}
Total:     ${total:>9,.2f}
"""
print(output)
```

```
Subtotal:  $ 1,598.40
Sales Tax: $   103.90
Total:     $ 1,702.30
```

FIGURE 1-10: All dollar amounts are right aligned within a width of 9 characters (>9).

You may look at Figure 1-10 and wonder why the dollar signs are lined up the way they are. Why aren't they aligned right next to their numbers? The dollar signs are part of the literal string, outside the curly braces, so they aren't affected by the >9 inside the curly braces.

Realigning the dollar signs is a little more complicated than you might imagine, because you can use the `,.2f` formatting only on a number. You can't attach a `$` to the front of a number unless you change the number to a string — but then it wouldn't be a number anymore, so `.2f` wouldn't work.

But complicated doesn't mean impossible; it just means inconvenient. You can convert each dollar amount to a string in the current format, stick the dollar sign on that string, and then format the width and alignment on this string. For example, the following code creates a variable named `s_subtotal` containing a dollar sign immediately followed by the dollar amount, with the dollar sign just to the left of the first digit and no spaces after the dollar sign:

```
s_subtotal = "$" + f"{subtotal:,.2f}"
```

In this code, we assume the `subtotal` variable contains some number. Let's say the number is `1598.402`, though it could be any number. The `f"{subtotal:,2f}"` formats the number in a fixed two-decimal-places format with a comma in the thousands place, like this:

```
1,598.40
```

The output is a string rather than a number because an f-string always produces a string.

The following part of the code sticks (concatenates) a dollar sign in the front:

```
"$"+
```

So now the output is `$1,598.40`. That final formatted string is stored in a new variable named `s_subtotal`. (We added the leading `s_` to remind us that this is the string equivalent of the subtotal number, not the original number.)

To display that dollar amount right aligned with a width of 9 digits, use `>9` in a new format string to display the `s_subtotal` variable, like this:

```
f{s_subtotal:>9}
```

REMEMBER

When you use `+` with strings, you *concatenate* (join) the two strings. The `+` only does addition with numbers, not strings.

Figure 1-11 shows a complete example, including the output from running the code. All the numbers are right aligned with the dollar signs in the usual place.

```
# Numerical values
unit_price = 49.95
quantity = 32
sales_tax_rate = 0.065
subtotal = quantity * unit_price
sales_tax = sales_tax_rate * subtotal
total = subtotal + sales_tax

# Format amounts to show as string with leading dollar sign
s_subtotal = "$" + f"{subtotal:,.2f}"
s_sales_tax = "$" + f"{sales_tax:,.2f}"
s_total = "$" + f"{total:,.2f}"

# Output the string with dollar sign already attached
output=f"""
Subtotal:  {s_subtotal:>9}
Sales Tax: {s_sales_tax:>9}
Subtotal:  {s_total:>9}
"""
print(output)
```

```
Subtotal:  $1,598.40
Sales Tax:   $103.90
Subtotal:  $1,702.30
```

FIGURE 1-11: All the dollar amounts neatly aligned.

Grappling with Weirder Numbers

Most of us deal with simple numbers like quantities and dollar amounts all the time. If your work requires you to deal with bases other than 10 or imaginary numbers, Python has the stuff you need to do the job. But keep in mind that you don't need to learn these things to use Python or any other language. You would use these only if your actual work (or perhaps homework) requires it. In the next section, you look at some number types commonly used in computer science: binary, octal, and hexadecimal numbers.

Binary, octal, and hexadecimal numbers

If your work requires dealing with base 2, base 8, or base 16 numbers, you're in luck because Python has symbols for writing these as well as functions for converting among them. Table 1-3 shows the three non-decimal bases and the digits used by each.

TABLE 1-3 Python for Base 2, 8, and 16 Numbers

System	Also Called	Digits Used	Symbol	Function
Base 2	Binary	0,1	0b	`bin()`
Base 8	Octal	0,1,2,3,4,5,6,7	0o	`oct()`
Base 16	Hexadecimal or hex	0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F	0x	`hex()`

Most people never have to work with binary, octal, or hexadecimal numbers. So if all of this is giving you the heebie-jeebies, don't sweat it. If you've never heard of them before, chances are you'll never hear of them again after you've completed this section.

TIP

If you want more information about the various numbering systems, you can use your favorite search engine to search for *binary number* or *octal*, or *decimal*, or *hexadecimal*.

You can use these various functions to convert how the number is displayed at the Python prompt, of course, as well as in an apps you create. At the prompt, just use the `print()` function with the conversion function inside the parentheses, and the number you want to convert inside the innermost parentheses. For example, the following displays the hexadecimal equivalent of the number 255:

```
print(hex(255))
```

The result is `0xff`, where the `0x` indicates that the number that follows is expressed in hex, and `ff` is the hexadecimal equivalent of 255.

To convert from binary, octal, or hex to decimal, you don't need to use a function. Just use `print()` with the number you want to convert inside the parentheses. For example, `print(0xff)` displays 255, the decimal equivalent of hex `ff`. Figure 1-12 shows some more examples you can try at the Python prompt.

```
x=255
# Convert decimal to other number systems
print(bin(x))
print(oct(x))
print(hex(x))

# Show number in decimal number system (no conversion required)
print(0b11111111)
print(0o377)
print(0xff)
```

```
0b11111111
0o377
0xff
255
255
255
```

FIGURE 1-12: Messing about with binary, octal, and hex.

Complex numbers

Complex numbers are another one of those weird numbering things you may never have to deal with unless you happen to be into electrical engineering, higher math, or a branch of science that uses them. A *complex number* is one that can be expressed as $a+bi$ where a and b are real numbers, and i represents the imaginary

number satisfied by the equation $x^2 = -1$. There is no real number x whose square equals -1, so that's why it's called an *imaginary number*.

Some branches of math use a lowercase *i* to indicate an imaginary number. But Python uses *j* (as do those in electrical engineering because *i* is used to indicate current).

Anyway, if your application requires working with complex numbers, you can use the `complex()` function to generate an imaginary number, using the following syntax:

```
complex(real,imaginary)
```

Replace *`real`* with the real part of the complex number, and replace *`imaginary`* with the imaginary number. For example, in code or the command prompt, try this:

```
z = complex(2,-3)
```

The variable `z` gets the imaginary number `2-3j`. Then use a `print()` function to display the contents of `z`, like this:

```
print(z)
```

The screen displays the imaginary number `2-3j`.

You can tack `.real` or `.imag` onto an imaginary number to get the real or imaginary part. For example, the following produces `2.0`, which is the real part of the number `z`:

```
print(z.real)
```

And this returns `-3.0`, which is the imaginary part of `z`:

```
print(z.imag)
```

Once again, if none of this makes sense to you, don't worry. It's not required for learning or doing Python. Python simply offers complex numbers and these functions for people who happen to require them.

If your work requires working with complex numbers, search the web for *python cmath* to learn about Python's `cmath` module, which provides functions for complex numbers.

Manipulating Strings

In Python and other programming languages, we refer to words and chunks of text as *strings,* short for "a string of characters." A string has no numeric meaning or value. (We discuss the basics of strings in Book 1, Chapter 4.) In this section, you learn Python coding skills for working with strings.

Concatenating strings

You can join strings by using a + sign. The process of doing so is called *string concatenation* in nerd-o-rama world. One thing that catches beginners off-guard is the fact that a computer doesn't know a word from a bologna sandwich. So when you join strings, the computer doesn't automatically put spaces where you'd expect them. For example, in the following code, `full_name` is a concatenation of the first three strings.

```
first_name = "Alan"
middle_init = "C"
last_name = "Simpson"
full_name = first_name+middle_init + last_name
print(full_name)
```

When you run this code to print the contents of the `full_name` variable, you can see that Python did join them in one long string:

```
AlanCSimpson
```

Nothing is wrong with this output, per se, except that we usually put spaces between words and the parts of a person's name.

Because Python won't automatically put in spaces where you think they should go, you have to put them in yourself. The easiest way to represent a single space is by using a pair of quotation marks with one space between them, like this:

```
" "
```

If you forget to put the space between the quotation marks, like the following, you won't get a space in your string because there's no space between the quotation marks:

```
""
```

You can put multiple spaces between the quotation marks if you want multiple spaces in your output, but typically one space is enough. In the following example, you put a space between `first_name` and `last_name`. You also stick a period and space after `middle_init`:

```
first_name = "Alan"
middle_init = "C"
last_name = "Simpson"
full_name = first_name + " " + middle_init + ". " + last_name
print(full_name)
```

The output of this code, which is the contents of that `full_name` variable, looks more like the kind of name you're used to seeing:

```
Alan C. Simpson
```

Getting the length of a string

To determine how many characters are in a string, you use the built-in `len()` function (short for *length*). The length includes spaces because spaces are characters, each one having a length of one. An empty string — that is, a string with nothing in it, not even a space — has a length of zero.

Here are some examples. In the first line you define a variable named `s1` and put an empty string in it (a pair of quotation marks with nothing in between). The `s2` variable gets a space (a pair of quotation marks with a space between). The `s3` variable gets a string with some letters and spaces. Then, three `print()` functions display the length of each string:

```
s1 = ""
s2 = " "
s3 = "A B C"
print(len(s1))
print(len(s2))
print(len(s3))
```

Following is the output from that code, when executed. The output makes perfect sense when you understand that `len()` measures the length of strings as the number of characters (including spaces) in the string:

```
0
1
5
```

Working with common string operators

Python offers several operators for working with sequences of data. One weird thing about strings in Python (and in most other programming languages) is that when you're counting characters, the first character counts as 0, not 1. This makes no sense to us humans. But computers count characters that way because it's the most efficient method. So even though the string in Figure 1-13 is five characters long, the last character in that string is the number 4, because the first character is number 0. Go figure.

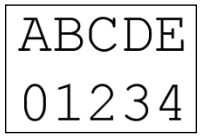


FIGURE 1-13: Character positions in a string start at 0, not 1.

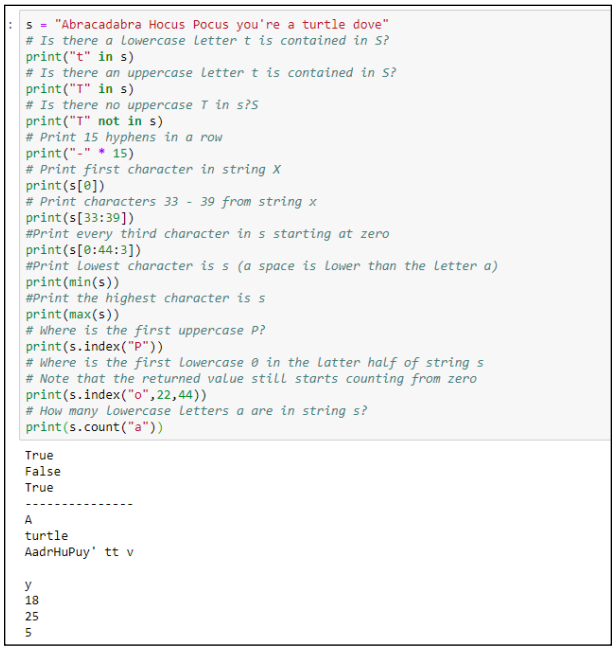
Table 1-4 summarizes the Python 3 operators for working with strings.

TABLE 1-4 Python Sequence Operators That Work with Strings

Operator	Purpose
`x in s`	Returns `True` if `x` exists somewhere in string `s`.
`x not in s`	Returns `True` if `x` is not contained in string `s`.
`s * n or n * s`	Repeats string `s` `n` times.
`s[i]`	The `i` th item of string `s` where the first character is `0`.
`s[i:j]`	A slice from string `x` beginning with the character at position `i` through to the character at position `j`.
`s[i:j:k]`	A slice of `s` from `i` to `j` with step `k`.
`min(s)`	The smallest (lowest) character of string `s`.
`max(s)`	The largest (highest) character of string `s`.
`s.index(x[, i[, j]])`	The numeric position of the first occurrence of `x` in string `s`. The optional `i` and `j` limit the search to the characters from `i` to `j`.
`s.count(x)`	The number of times string `x` appears in larger string `s`.

Figure 1-14 shows examples of using the string operators in Jupyter Notebook. When the output of a `print()` function doesn't look right, keep in mind two important facts about strings in Python:

- The first character is always number `0`.
- Every space counts as one character, so don't skip spaces when counting.



```
s = "Abracadabra Hocus Pocus you're a turtle dove"
# Is there a lowercase letter t is contained in S?
print("t" in s)
# Is there an uppercase letter t is contained in S?
print("T" in s)
# Is there no uppercase T in s?S
print("T" not in s)
# Print 15 hyphens in a row
print("-" * 15)
# Print first character in string X
print(s[0])
# Print characters 33 - 39 from string x
print(s[33:39])
#Print every third character in s starting at zero
print(s[0:44:3])
#Print lowest character is s (a space is lower than the letter a)
print(min(s))
#Print the highest character is s
print(max(s))
# Where is the first uppercase P?
print(s.index("P"))
# Where is the first lowercase 0 in the latter half of string s
# Note that the returned value still starts counting from zero
print(s.index("o",22,44))
# How many lowercase letters a are in string s?
print(s.count("a"))
```

```
True
False
True
---------------
A
turtle
AadrHuPuy' tt v

y
18
25
5
```

FIGURE 1-14: Playing around with string operators in Jupyter Notebook.

You may have noticed that `min(s)` returns a blank space, meaning that the blank space character is the lowest character in that string. But what exactly makes the space "lower" than the letter *A* or the letter *a*? The simple answer is the letter's *ASCII number*. Every character you can type at your keyboard, and many additional characters, have a number assigned by the American Standard Code for Information Interchange (ASCII).

Figure 1-15 shows a chart with ASCII numbers for many common characters. Spaces and punctuation characters are "lower" than *A* because they have smaller ASCII numbers. Uppercase letters are "lower" than lowercase letters because they have smaller ASCII numbers. Are you wondering what happened to the characters assigned to numbers 0–31? These numbers have characters too, but they are control characters and are essentially non-printing and invisible, such as when you hold down the Ctrl key and press another key.

Number	Character	Number	Character	Number	Character
32	[space]	65	A	97	a
33	!	66	B	98	b
34	"	67	C	99	c
35	#	68	D	100	d
36	$	69	E	101	e
37	%	70	F	102	f
38	&	71	G	103	g
39	'	72	H	104	h
40	(	73	I	105	i
41	)	74	J	106	j
42	*	75	K	107	k
43	+	76	L	108	l
44	,	77	M	109	m
45	-	78	N	110	n
46	.	79	O	111	o
47	/	80	P	112	p
48	0	81	Q	113	q
49	1	82	R	114	r
50	2	83	S	115	s
51	3	84	T	116	t
52	4	85	U	117	u
53	5	86	V	118	v
54	6	87	W	119	w
55	7	88	X	120	x
56	8	89	Y	121	y
57	9	90	Z	122	z
58	:	91	[	123	{
59	;	92	\	124	|
60	<	93	]	125	}
61	=	94	^	126	~
62	>	95	_	127	□
63	?	96	`	128	€
64	@				

FIGURE 1-15: ASCII numbers for common characters.

Python offers two functions for working with ASCII. The `ord()` function takes a character as input and returns the ASCII number of that character. For example, `print(ord("A"))` returns 65, because an uppercase *A* is character 65 in the ASCII chart. The `chr()` function does the opposite. You give it a number, and it returns the ASCII character for that number. For example, `print(chr(65))` displays `A` because *A* is character 65 in the ASCII chart.

Manipulating strings with methods

Every string in Python 3 is considered a *str object* (pronounced "string object"). The shortened word *str* for *string* distinguishes Python 3 from earlier versions of Python, which referred to string as string objects (with the word *string* spelled out, not shortened). This naming convention is a great source of confusion, especially for beginners. Just try to remember that in Python 3, str is all about strings of characters.

Python offers numerous *str methods* (also called *string methods*) to help you work with str objects. The general syntax of str object methods is as follows:

```
string.methodname(params)
```

where *string* is the string you're analyzing, *methodname* is the name of a method from Table 1-5, and *params* refers to any parameters that you need to pass to the method (if required). The leading `s` in the first column of Table 1-5 means "any string," be it a literal string enclosed in quotation marks or the name of a variable that contains a string.

TABLE 1-5 Built-In Methods for Python 3 Strings

Method	Purpose
`s.capitalize()`	Returns a string with the first letter capitalized and the rest lowercase.
`s.count(x, [y, z])`	Returns the number of times string `x` appears in string `s`. Optionally, you can add `y` as a starting point and `z` as an ending point to search a portion of the string.
`s.find(x, [y, z])`	Returns a number indicating the first position at which string `x` can be found in string `s`. Optional *y* and *z* parameters allow you to limit the search to a portion of the string. Returns –1 if none found.
`s.index(x, [y, z])`	Similar to `find` but returns a "substring not found" error if string `x` can't be found in string `y`.
`s.isalpha()`	Returns `True` if `s` is at least one character long and contains only letters (A-Z or a-z).
`s.isdecimal()`	Returns `True` if `s` is at least one character long and contains only numeric characters (0-9).
`s.islower()`	Returns `True` if `s` contains letters and all those letters are lowercase.
`s.isnumeric()`	Returns `True` if `s` is at least one character long and contains only numeric characters (0-9).
`s.isprintable()`	Returns `True` if string `s` contains only printable characters.
`s.istitle()`	Returns `True` if string `s` contains letters and the first letter of each word is uppercase followed by lowercase letters.
`s.isupper()`	Returns `True` if all letters in the string are uppercase.
`s.lower()`	Returns `s` with all letters converted to lowercase.
`s.lstrip()`	Returns `s` with any leading spaces removed.
`s.replace(x, y)`	Returns a copy of string `s` with all characters `x` replaced by character `y`.
`s.rfind(x, [y, z])`	Similar to `s.find` but searches backward from the start of the string. If `y` and `z` are provided, searches backward from position `z` to position `y`. Returns –1 if string `x` not found.

Method	Purpose
`s.rindex()`	Same as `s.rfind` but returns an error if the substring isn't found.
`s.rstrip()`	Returns string `x` with any trailing spaces removed.
`s.strip()`	Returns string `x` with leading and trailing spaces removed.
`s.swapcase()`	Returns string `s` with uppercase letters converted to lowercase and lowercase letters converted to uppercase.
`s.title()`	Returns string `s` with the first letter of every word capitalized and all other letters lowercase.
`s.upper()`	Returns string `s` with all letters converted to uppercase.

You can play around with these methods in a Jupyter notebook, at the Python prompt, or in a `.py` file. Figure 1-16 shows some examples in a Jupyter notebook using three variables named `s1`, `s2`, and `s3` as strings to experiment with. The result of running the code appears below the code.

```
s1 = "There is no such word as schmeedledorp"
s2="   a b c   "
s3="ABC"
# Captialize first letter, the rest lowercase
print(s3.capitalize())
# Count the number of spaces in s1
print(s1.count(" "))
# Find the dot in S4
print(s4.find("."))
# Is s2 all lowercase letters?
print(s2.islower())
# Convert s3 to all lowercase
print(s3.lower())
# String leading characters from s2
print(s2.lstrip())
# String leading and trailing characters from s2
print(s2.strip())
# Swap the case of letters in s1
print(s1.swapcase())
# Show s1 in title case (initial caps)
print(s1.title())
# Show s1 uppercase
print(s1.upper())
```

```
Abc
6
3
True
abc
a b c
a b c
tHERE IS NO SUCH WORD AS SCHMEEDLEDORP
There Is No Such Word As Schmeedledorp
THERE IS NO SUCH WORD AS SCHMEEDLEDORP
```

FIGURE 1-16: Playing around with Python 3 string functions.

REMEMBER

Don't bother trying to memorize or even make sense of every string method. Remember instead that if you need to operate on a string in Python, you can do a web search for *python 3 string methods* to find out what's available.

Uncovering Dates and Times

In the world of computers, we often use dates and times for scheduling, or for calculating when something is due or how many days it's past due. We sometimes use *timestamps* to record exactly when a user did something or when an event occurred. There are lots of reasons for using dates and times in Python, but perhaps surprisingly, no built-in data type for them exists like the ones for strings and numbers.

To work with dates and times, you typically need to use the `datetime` module. Like any module, you must import it before you can use it. You do that using `import datetime`. As with any import, you can add an alias (nickname) that's easier to type, if you like. For example, `import datetime as dt` would work too. You just have to remember to type `dt` rather than `datetime` in your code when calling upon the capabilities of that module.

The `datetime` module is an abstract base class, which is a fancy way of saying it offers new data types to the language. For dates and times, those data types are as follows:

- **`datetime.date`:** A date consisting of month, day, and year (but no time information).
- **`datetime.time`:** A time consisting of hour, minute, second, microsecond, and optionally time zone information if needed (but no date).
- **`datetime.datetime`:** A single item of data consisting of date, time, and optionally time zone information.

We preceded each type with the full word `datetime` in the preceding list, but if you use an alias, such as `dt`, you can use that in your code instead. We talk about each of these data types separately in the sections that follow.

Working with dates

The `datetime.date` data type is ideal for working with dates when time isn't an issue. You can create a date object in two ways. You can get today's date from the computer's internal clock by using the `today()` method. Or you can specify a year, month, and day (in that order) inside parentheses.


REMEMBER

When specifying the month or day, never use a leading zero for `datetime.date()`. For example, April 1 2020 has to be expressed as `2020,4,1` — if you type `2020,04,01`, it won't work.

For example, after importing the `datetime` module, you can use `date.today()` to get the current date from the computer's internal clock. Or use `date(`*`year`*`,` *`month`*`,` *`day`*`)` syntax to create a date object for some other date. The following code shows both methods:

```
# Import the datetime module, nickname dt
import datetime as dt
# Store today's date in a variable named today.
today = dt.date.today()
# Store some other date in a variable called last_of_teens
last_of_teens = dt.date(2019, 12, 31)
```

Try it by typing the code in a Jupyter notebook, at the Python prompt, or in a `.py` file. Use the `print()` function to see what's in each variable, as shown in Figure 1-17. Your `today` variable won't be the same as in the figure; it will the date you try this.

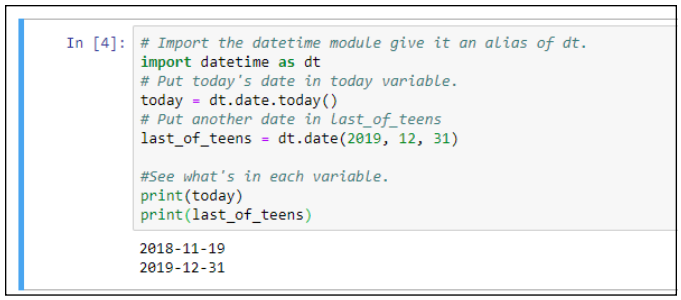


FIGURE 1-17: Experiments with `datetime.date` objects in a Jupyter notebook.

YOUR COMPUTER DATE AND TIME

If your computer is connected to the Internet, its internal date and time should be accurate. That's because it gets that information from NNTP (Network News Transfer Protocol), a standard time that any computer or app can get from the Internet.

The datetime information is tailored to your time zone and takes into account the daylight saving time of your location (if applicable). So in other words, the date and time shown on your computer screen should match what the calendar and the clock on your wall say.

You can isolate any part of a date object by using `.month`, `.day`, or `.year`. For example, in the same Jupyter cell or Python prompt, execute this code:

```
print(last_of_teens.month)
print(last_of_teens.day)
print(last_of_teens.year)
```

Each of the three components of that date appear on a separate line:

```
12
31
2019
```

As you saw on the first printout, the default date display is *yyyy-mm-dd*, but you can format dates and times however you want. Use f-strings, which we discuss earlier in this chapter, along with the directives shown in Table 1-6, which includes the format for dates as well as for times, as we discuss later in this chapter.

TABLE 1-6 Formatting Strings for Dates and Times

Directive	Description	Example
%a	Weekday, abbreviated	Sun
%A	Weekday, full	Sunday
%w	Weekday number 0-6, where 0 is Sunday	0
%d	Number day of the month 01-31	31
%b	Month name abbreviated	Jan
%B	Month name full	January
%m	Month number 01-12	01
%y	Year without century	19
%Y	Year with century	2019
%H	Hour 00-23	23
%I	Hour 00-12	11
%p	AM/PM	PM
%M	Minute 00-59	01
%S	Second 00-59	01

Directive	Description	Example
%f	Microsecond 000000-999999	495846
%z	UTC offset	-0500
%Z	Time zone	EST
%j	Day number of year 001-366	300
%U	Week number of year, Sunday as the first day of week, 00-53	50
%W	Week number of year, Monday as the first day of week, 00-53	50
%c	Local version of date and time	Tue Dec 31 23:59:59 2018
%x	Local version of date	12/31/18
%X	Local version of time	23:59:59
%%	A % character	%

Some tutorials tell you to format dates and times by using `strftime` rather than f-strings, and that's certainly a valid method. We're sticking with the newer f-strings here, however, because we think they'll be preferred over `strftime` in the future.

When using format strings, make sure you put spaces, slashes, and anything else you want between directives where you want those to appear in the output. For example, this line:

```
print(f"{last_of_teens:%A, %B %d, %Y}")
```

when executed, displays this:

```
Tuesday, December 31, 2019
```

To show the date in the *mm/dd/yyyy* format, use `%m/%d/%Y`, like this:

```
todays_date = f"{today:%m/%d/%Y}"
```

The output will be the current date for you when you try it, with a format like the following:

```
11/19/2018
```

Table 1-7 shows a few more examples you can try with different dates.

TABLE 1-7 Sample Date Format Strings

Format String	Example
%a, %b %d %Y	Sat, Jun 01 2019
%x	06/01/19
%m-%d-%y	06-01-19
This %A %B %d	This Saturday June 01
%A %B %d is day number %j of %Y	Saturday June 01 is day number 152 of 2019

Working with times

If you want to work strictly with time data, use the `datetime.time` class. The basic syntax for defining a time object using the `time` class is

```
variable = datetime.time([hour,[minute,[second,[microsecond]]]])
```

Notice how all the arguments are optional. For example, you can use no arguments:

```
midnight = dt.time()
print(midnight)
```

This code stores the time as 00:00:00, which is midnight. To verify that it's really a time, entering `print(type(midnight))` displays the following:

```
00:00:00
<class 'datetime.time'>
```

The second line tells you that the 00:00:00 value is a `time` object from the `datetime` class.

The fourth optional value you can pass to `time()` is `microseconds` (millionths of a second). For example, the following code puts a time that's a millionth of a second before midnight in a variable named `almost_midnight` and then displays that time onscreen with a `print()` function:

```
almost_midnight = dt.time(23, 59, 59, 999999)
print(almost_midnight)
23:59:59.999999
```

You can use format strings with the time directives from Table 1-6 to control the format of the time. Table 1-8 shows some examples using 23:59:59:999999 as the sample time.

TABLE 1-8 Sample Date Format Strings

Format String	Example
%I:%M %p	11:59 PM
%H:%M:%S and %f microseconds	23:59:59 and 999999 microseconds
%X	23:59:59

Sometimes you want to work only with dates, and sometimes you want to work only with times. Often you want to pinpoint a moment in time using both the date and the time. For that, use the `datetime` class of the `datetime` module. This class supports a `now()` method that can grab the current date and time from the computer clock, as follows:

```
import datetime as dt
right_now = dt.datetime.now()
print(right_now)
```

What you see on the screen from the `print()` function depends on when you execute this code. But the format of the `datetime` value will be like this:

```
2019-11-19 14:03:07.525975
```

This means November 19, 2019 at 2:03 PM (with 7.525975 seconds tacked on).

You can also define a datetime using any the following parameters. The month, day, and year are required. The rest are optional and set to 0 in the time if you omit them.

```
datetime(year, month, day, hour, [minute, [second, [microsecond]]])
```

Here is an example using 11:59 PM on December 31 2019:

```
import datetime as dt
new_years_eve = dt.datetime(2019, 12, 31, 23, 59)
print(new_years_eve)
```

Here is the output of that `print()` statement with no formatting:

```
2019-12-31 23:59:00
```

Table 1-9 shows examples of formatting the datetime using directives shown previously in Table 1-6.

TABLE 1-9 Sample Datetime Format Strings

Format String	Example
%A, %B %d at %I:%M%p	Tuesday, December 31 at 11:59PM
%m/%d/%y at %H:%M%p	12/31/19 at 23:59
%I:%M %p on %b %d	11:59 PM on Dec 31
%x	12/31/19
%c	Tue Dec 31 23:59:00 2019
%m/%d/%y at %I:%M %p	12/31/19 at 11:59 PM
%I:%M %p on %m/%d/%y	1:59 PM on 12/31/2019

Calculating timespans

Sometimes just knowing the date or time isn't enough. You need to know the duration, or *timespan*, as it's typically called in the computer world. In other words, not the date, not the o'clock, but the "how long" in terms of years, months, weeks, days, hours, minutes, or whatever. For timespans, the Python `datetime` module includes the `datetime.timedelta` class.

A `timedelta` object is created automatically whenever you subtract two dates, times, or datetimes to determine the duration between them. For example, suppose you create a couple of variables to store dates, perhaps one for New Year's Day and another for Memorial Day. Then you create a third variable named `days_between` and put in it the difference you get by subtracting the earlier date from the later date, as follows:

```
import datetime as dt
new_years_day = dt.date(2019, 1, 1)
memorial_day = dt.date(2019, 5, 27)
days_between = memorial_day - new_years_day
```

So what exactly is `days_between` in terms of a data type? If you print its value, you get `146 days, 0:00:00`. In other words, there are 146 days between those dates; the `0:00:00` is time but because we didn't specify a time of day in either date, the time digits are all just set to `0`. If you use the Python `type()` function to determine the data type of `days_between`, you see it's a `timedelta` object from the `datetime` class, as follows:

```
146 days, 0:00:00
<class 'datetime.timedelta'>
```

The `timedelta` calculation happens automatically when you subtract one date from another to get the time between. You can also define any `timedelta` (duration) using this syntax:

```
datetime.timedelta(days=, seconds=, microseconds=, milliseconds=, minutes=,
   hours=, weeks=)
```

If you provide an argument, you must include a number after the = sign. If you omit an argument, its value is set to 0.

To get an understanding of how this works, try out the following code. After importing the `datetime` module, create a date using `.date()`. Then create a `timedelta` object using `.timedelta`. If you add a `date` and a `timedelta`, you get a new date — in this case, a date that's 146 days after 1/1/2019:

```
import datetime as dt
new_years_day = dt.date(2019, 1, 1)
duration = dt.timedelta(days=146)
print(new_years_day + duration)
2019-05-27
```

Of course, you can subtract too. For example, if you start with a date of 5/27/2019 and subtract 146 days, you get 1/1/2019, as shown here:

```
import datetime as dt
memorial_day = dt.date(2019, 5, 27)
duration = dt.timedelta(days=146)
print(memorial_day - duration)
2019-01-01
```

It works with datetimes too. If you're looking for a duration that's less than a day, just give both times the same date. For example, consider the following code and the results of the subtraction:

```
import datetime as dt
start_time = dt.datetime(2019, 3, 31, 8, 0, 0)
```

```
finish_time = dt.datetime(2019, 3, 31, 14, 34, 45)
time_between = finish_time - start_time
print(time_between)
print(type(time_between))
```

```
6:34:45
<class 'datetime.timedelta'>
```

We know that 6:34:45 is a time duration of 6 hours 34 minutes and 45 seconds for two reasons. One, it's the result of subtracting one moment of time from another. Two, printing the `type()` of that data type tells us it's a `timedelta` object (a duration), not an o'clock time.

Here is another example using datetimes with different dates: One is the current datetime, and the other is a date of birth with the time down to the minute (March 31 1995 at 8:26 AM). To calculate age, subtract the birthdate from the current time, `now`:

```
import datetime as dt
now = dt.datetime.now()
birthdatetime = dt.datetime(1995, 3, 31, 8, 26)
age = now - birthdatetime
print(age)
print(type(age))
8634 days, 7:55:07.739804
<class 'datetime.timedelta'>
```

The result is expressed as follows:

```
8634 days, 7 hours, 52 minutes, and 1.967031 seconds
```

The tiny seconds value stems from the fact that `datetime.now` grabs the date and time from the computer's clock down to the microsecond.

You don't always need microseconds or even seconds in your `timedelta` object. For example, say you're trying to determine someone's age. You could start by creating two dates, one named `today` for today's date and another named `birthdate` that contains the birthdate. The following example uses a birthdate of Jan 31, 2000:

```
import datetime as dt
today = dt.date.today()
birthdate = dt.date(2000, 12, 31)
```

```
delta_age = (today - birthdate)
print(delta_age)
```

The last two lines create a variable named `delta_age` and print what's in the variable. If you run this code, you'll see something like the following output (but it won't be exactly the same because your `today` date will be whatever today's date is when you run the app):

```
6533 days, 0:00:00
```

Let's say what we really want is the age in years. You can convert `timedelta` to a number of days by tacking `.days` onto `timedelta`. You can put that in another variable called `days_old`. Printing `days_old` and its type show you that `days_old` is an `int`, a regular old integer you can do math with. For example, in the following code, the `days_old` variable receives the value `delta_age.days`, which is `delta_age` from the preceding line converted to a number of days:

```
delta_age = (today - birthdate)
days_old = delta_age.days
print(days_old, type(days_old))
6533 <class 'int'>
```

To get the number of years, divide the number of days by 365. If you want just the number of years as an integer, use the floor division operator (`//`) rather than regular division (`/`). (*Floor division* removes the decimal portion from the quotient, so you get a whole number). You can put the result of that calculation in another variable if you like. For example, in the following code, the `years_old` variable contains a value calculated by dividing `days_old` by `365`:

```
years_old = days_old // 365
print(years_old)
18
```

So we get the age, in years: 18. If you want the number of months, too, you can ballpark that just by taking the remainder of dividing the days by 365 to get the number of days left. Then floor divide that value by 30 (because on average each month has about 30 days) to get a good approximation of the number of months. Use `%` rather than `/` for division to get just the remainder after the division. Figure 1-18 shows the sequence of events in a Jupyter notebook, with comments to explain what's going on.

```
import datetime as dt
# Today's date according to your computer
today = dt.date.today()

# Any birthdate expressed as year, month, day
birthdate = dt.date(2000, 1, 31)

# Duration between the dates as a timedelta
delta_age = (today - birthdate)

# Duration between the dates as a number (of days)
days_old = delta_age.days

# Floor divide days by 365 to get the number of years
years = days_old // 365

# Days left over is remainder of days_old divided by 365.
# Floor divide that remainder by 30 for approximate months.
months = (days_old % 365) // 30

# Print in a format to your liking
print(f"You are {years} years and {months} months old.")
```

```
You are 18 years and 9 months old.
```

FIGURE 1-18: Calculating age in years and months from a `timedelta` object.

Accounting for Time Zones

As you know, when it's noon in your neighborhood, it doesn't mean its noon everywhere. Figure 1-19 shows a map of all the time zones. If you want a closer look, simply search the web for *time zone map*. At any given moment, it's a different day and time of day depending on where you happen to be on the globe. There is a universal time, called the Coordinated Universal Time or Universal Time Coordinated (UTC). You may have heard of Greenwich Mean Time (GMT) or Zulu time used by the military, which is the same idea. All these times refer to the time at the prime meridian on Earth, or 0 degrees longitude, smack dab in the middle of the time zone map in Figure 1-19.

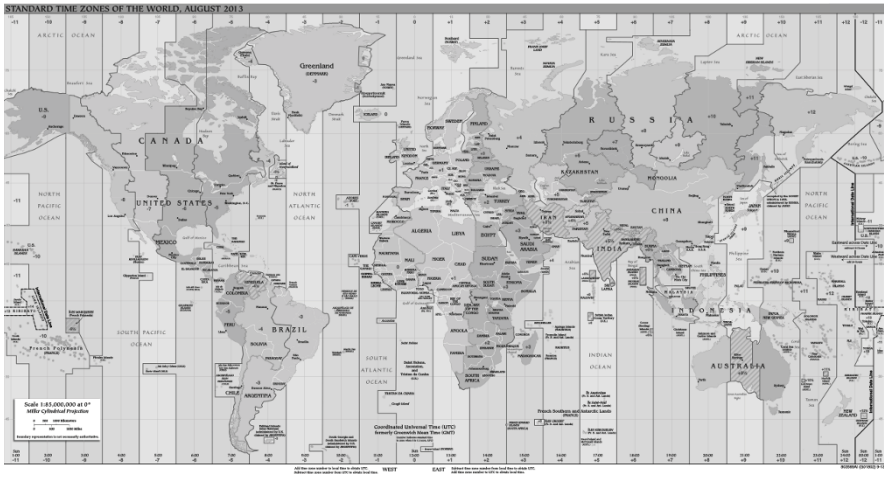


FIGURE 1-19: Time zones.

These days, most people rely on the Olson Database as the primary source of information about time zones. It lists all current time zones and locations. Do a web search for *Olson database* or *tz database* if you're interested in all the details. There are too many time zone names to list here, but Table 1-10 shows some examples of American time zones. The left column is the official name from the database. The second column shows the more familiar name. The last two columns show the offset from UTC for standard time and daylight saving time.

TABLE 1-10 Sample Time Zones from the Olson Database

Time Zone	Common Name	UTC Offset	UTC DST Offset
Etc/UTC	UTC	+00:00	+00:00
Etc/UTC	Universal	+00:00	+00:00
America/Anchorage	US/Alaska	–09:00	–08:00
America/Adak	US/Aleutian	–10:00	–09:00
America/Phoenix	US/Arizona	–07:00	–07:00
America/Chicago	US/Central	–06:00	–05:00
America/New_York	US/Eastern	–05:00	–04:00
America/Indiana/Indianapolis	US/East-Indiana	–05:00	–04:00
America/Honolulu	US/Hawaii	–10:00	–10:00
America/Indiana/Knox	US/Indiana-Starke	–06:00	–05:00
America/Detroit	US/Michigan	–05:00	–04:00
America/Denver	US/Mountain	–07:00	–06:00
America/Los Angeles	US/Pacific	–08:00	–07:00
Pacific/Pago_Pago	US/Samoa	–11:00	–11:00
Etc/UTC	UTC	+00:00	+00:00
Etc/UTC	Zulu	+00:00	+00:00

So why are we telling you all this? Because Python lets you work with two different types of datetimes:

- **Naïve datetime:** Any datetime that does not include information that relates it to a specific time zone
- **Aware datetime:** A datetime that includes time zone information

The `timedelta` objects and dates that you define with `.date()` are always naïve. Any time or datetime you create as `time()` or `datetime()` objects will also by naïve, by default. But with those two you have the option of including time zone information if it's useful in your work, such as when you're showing event dates to an audience in multiple time zones.

Working with Time Zones

When you get the time from your computer's system clock, it's for your time zone, but you don't have an indication of what that time zone *is*. But you can tell the difference between your time and UTC time by comparing `.now()` for your location to `.utc_now()` for UTC time, and then subtracting the difference, as shown in Figure 1-20.

```
# Get the dateime module and give it an alias
import datetime as dt

# Get the time from computer clock
here_now = dt.datetime.now()

# Get the UTC datetime right now
utc_now = dt.datetime.utcnow()

# Subtract to see difference
time_difference = (utc_now - here_now)

# Show results
print(f"My time   : {here_now:%I:%M %p}")
print(f"UTC time  : {utc_now:%I:%M %p}")
print(f"Difference: {time_difference}")
```

```
My time   : 01:02 PM
UTC time  : 06:02 PM
Difference: 5:00:00
```

FIGURE 1-20: Determining the difference between your time and UTC time.

When we ran that code, the current time was 1:02PM and the UTC time was 6:02PM. The difference was 5:00:00, which means five hours (no minutes or seconds). Our time is earlier, so our time zone is really UTC – 5 hours.

Note that if you subtract the earlier time from the later time, you get a negative number, which can be misleading, as follows:

```
time_difference = (here_now - utc_now)
Difference: -1 day, 19:00:00
```

That's still five hours, really, because if you subtract 1 day and 19 hours from 24 hours (one day), you still get 5 hours. Tricky business. But keep in mind the left

side of the time zone map is east, and the sun rises in the east in each time zone. So when it's rising in your time zone, it's already risen in time zones to the right, and hasn't yet risen in time zones to your left.

If you want to work directly with time zone names, you'll need to import some date utilities from Python's `dateutils` package. In particular, you need `gettz` (short for *get timezone*) from the `tz` class of `dateutil`. So in your code, right after the line where you `import datetime`, use `from dateutil.tz import gettz` like this:

```
# import datetime and dateutil tz
import datetime as dt
from dateutil.tz import gettz
```

Afterwards, you can use `gettz('name')` to get time zone information for any time zone. Replace *name* with the name of the time zone from the Olson database: for example, `America/New_York` for USA Eastern Time, or `Etc_UTC` for UTC Time.

Figure 1-21 shows an example where we get the current date and time using `datetime.now()` with five different time zones — UTC and four US time zones.

```
# import datetime, give it an alias
import datetime as dt
# import timezone helpers from dateutil
from dateutil.tz import gettz

# UTC time right now.
utc=dt.datetime.now(gettz('Etc/UTC'))
print(f"{utc:%A %D %I:%M %p %Z}")

# USA Eastern time.
est = dt.datetime.now(gettz('America/New_York'))
print(f"{est:%A %D %I:%M %p %Z}")

# USA Central time
cst=dt.datetime.now(gettz('America/Chicago'))
print(f"{cst:%A %D %I:%M %p %Z}")

# USA Mountain time
mst=dt.datetime.now(gettz('America/Boise'))
print(f"{mst:%A %D %I:%M %p %Z}")

pst=dt.datetime.now(gettz('America/Los_Angeles'))
print(f"{pst:%A %D %I:%M %p %Z}")
```

```
Friday 11/23/18 06:37 PM UTC
Friday 11/23/18 01:37 PM EST
Friday 11/23/18 12:37 PM CST
Friday 11/23/18 11:37 AM MST
Friday 11/23/18 10:37 AM PST
```

FIGURE 1-21: The current date and time for five different time zones.

All USA times are standard time because no one in the USA is on daylight saving time (DST) in late November. Let's see what happens if we schedule an event for some time in July, when the USA is on back on daylight saving time.

In this code (see Figure 1-22), we import `datetime` and `gettx` from `dateutil`, as we did in the preceding example. But we're not concerned about the current time. We're concerned about an event scheduled for July 4, 2020 at 7:00 PM in our local time zone. So we define that using the following:

```
event = dt.datetime(2020,7,4,19,0,0)
```

```
# import datetime and dateutil tz
import datetime as dt
from dateutil.tz import gettz

# July 4 Event, 7:00 local time (no specific time zone).
event = dt.datetime(2020,7,4,19,0,0)
# Show local date and time
print("Local: " + f"{event:%D %I:%M %p %Z}" + "\n")

event_eastern = event.astimezone(gettz("America/New_York"))
print(f"{event_eastern:%D %I:%M %p %Z}")

event_central = event.astimezone(gettz("America/Chicago"))
print(f"{event_central:%D %I:%M %p %Z}")

event_mountain = event.astimezone(gettz("America/Denver"))
print(f"{event_mountain:%D %I:%M %p %Z}")

event_pacific = event.astimezone(gettz("America/Los_Angeles"))
print(f"{event_pacific:%D %I:%M %p %Z}")

event_utc = event.astimezone(gettz("Etc/UTC"))
print(f"{event_utc:%D %I:%M %p %Z}")
```

```
Local: 07/04/20 07:00 PM

07/04/20 07:00 PM EDT
07/04/20 06:00 PM CDT
07/04/20 05:00 PM MDT
07/04/20 04:00 PM PDT
07/04/20 11:00 PM UTC
```

FIGURE 1-22: Date and time for a scheduled event in multiple time zones.

We didn't say anything about time zone in the date time, so the time will automatically be for our time zone. That datetime is stored in the `event` variable.

The following line of code (after the comment, which starts with `#`) shows the date and time, again local, because we didn't say anything about time zone. We added `"Local:"` to the start of the text, and added a line break at the end (`\n`) to visually separate that word from the rest of the output.

```
# Show local date and time
print("Local: " + f"{event:%D %I:%M %p %Z}" + "\n")
```

When the app runs, it displays the following output based on the datetime and our format string:

```
Local: 07/04/20 07:00 PM
```

The remaining code calculates the correct datetime for each of five time zones:

```
name = event.astimezone(gettz("tzname"))
```

The first *name* is just a variable name we made up. In `event.astimezone()`, the name `event` refers to the initial event time defined in a previous line. The `astimezone()` function is a built-in `dateutil` function that uses the following syntax:

```
.astimezone(gettz("tzname"))
```

In each line of code that calculates the date and time for a time zone, we replace *tzname* with the name of the time zone from the Olson database. As you can see in the output (refer to Figure 1-22), the datetime of the event for five different time zones is displayed. Note that the USA time zones are daylight saving time (such as EDT). Because we happen to be on the east coast and the event is in July, the correct local time zone is Eastern Daylight Time. When you look at the output of the dates, the first one matches our time zone, as it should, and the times for the remaining dates are adjusted for different time zones.

If you're thinking "Eek, what a complicated mess," you won't get any argument from us. None of this strikes us as intuitive, easy, or in the general ballpark of fun. But if you're in a pinch and need some time zone information for your data, the coding techniques you've learned so far should get you want you need.

If you research Python time zones online, you'll probably find that many people recommend using the `arrow` module rather than the `dateutil` module. We won't get into all that here, because `arrow` isn't part of your initial Python installation and this book is hefty enough. (If we tried to cover everything, you'd need a wheelbarrow to carry the book around.)

IN THIS CHAPTER

- Deciding with `if`
- Repeating with `for`
- Looping with `while`

Chapter 2
Controlling the Action

So far in this book we've talked a lot about storing information in computers, mostly in variables that Python and your computer can work with. Having the information in a form that the computer can work with is critical to getting a computer to do anything. Think of this as the "having" part — having some information with which to work.

But now we need to turn our attention to the "doing" part — working with that information to create something useful or entertaining. In this chapter, we cover the most important and most commonly used operations for making the computer *do* stuff. We start with something that computers do well, do quickly, and do a lot — make decisions.

Main Operators for Controlling the Action

You control what your program (and the computer) does by making decisions, which often involves making comparisons. You use operators, such as those in Table 2-1 to make comparisons. These operators are often referred to as *relational operators* or *comparison operators* because by comparing items the computer is determining how two items are related.

TABLE 2-1 Python Comparison Operators for Decision-Making

Operator	Meaning
==	Is equal to
!=	Is not equal to
<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to

Python also offers three *logical operators*, also called *Boolean operators*, which enable you assess multiple comparisons before making a final decision. These operators use the English word for, well, basically what they mean, as shown in Table 2-2.

TABLE 2-2 Python Logical Operators

Operator	Meaning
and	Both are true
or	One or the other is true
not	Is not true

In case you're wondering about that *Boolean* word, it's a reference to a guy named George Boole who, in the mid-1800s, helped establish the algebra of logic, which pretty much laid the foundation for today's computers. Feel free to do a web search for his name to learn more.

All these operators are often used with `if...then...else` decisions to control what an app or program does. To make such decisions, you use the Python `if` *statement.*

Making Decisions with if

The word *if* is used a lot in all apps and computer programs to make decisions. The simplest syntax for *if* follows:

```
if condition: do this
do this no matter what
```

So the first `do this` line is executed only if the condition is true. If the condition is false, that first `do this` is ignored. Regardless of what the condition turns out to be, the second line is executed next. Note that neither line is indented. Indentation means a lot in Python, as you'll see shortly. But first, let's do a few simple examples with this simple syntax. You can try it for yourself in a Jupyter notebook or `.py` file.

Figure 2-1 shows a simple example in which the `sun` variable receives the `down` string. Then an `if` statement checks to see whether the `sun` variable equals the word `down` and, if it does, prints a `Good night!` message. Then it just continues on normally to print an `I am here` message.

```
sun = "down"
if sun == "down": print("Good night!")
print("I am here")

   Good night!
   I am here
```

FIGURE 2-1: The result of a simple `if` when the condition proves true.

WARNING

Make sure you always use two equal signs with no space between (==) to test equality. This rule is easy to forget. If you type it incorrectly, the code won't work as expected.

If you run the same code with some word other than `down` in the `sun` variable, the first `print` is ignored. But the next line is executed normally because it's not dependent on the condition being true, as shown in Figure 2-2.

```
sun = "up"
if sun == "down": print("Good night!")
print("I am here")

   I am here
```

FIGURE 2-2: Result of simple `if` when the condition proves false.

In the second example, it's not true that the `sun` variable equals `down`; therefore the rest of that line is ignored and only the next line is executed.

In these two examples, the code to be executed when the condition proves true is on the same line as the if. However, often you want to do more than one thing when the condition proves true. For that, you'll need to indent each line to be executed only if the condition proves true. And code that's not indented below the `if` is executed whether the condition proves true or not. The recommendation is to indent by four spaces, but that's not a hard-and-fast rule. You just have to remember that each line has to be indented the same amount.

Also, you can use the indented syntax even if only one line of code is to be executed should the condition prove true. In fact, that's the most common way to write an `if` in Python because most people agree it makes the code more readable from a human perspective. So really, the syntax is

```
if condition:
    do this
     ...
do this no matter what
```

So if the condition proves true, the `do this` line is executed as are any other lines indented equally to that one. The first un-indented line under the `if` is executed no matter what. So you could write the simple `sun` example like this:

```
sun = "down"
if sun == "down":
    print("Good night!")
print("I am here")
```

As you can see in Figure 2-3, this code works the same as putting the code on one line. If `sun` is `down`, `Good night!` prints before the second `print` is executed. If `sun` doesn't equal `down`, the `print` statement for `Good night!` is skipped.

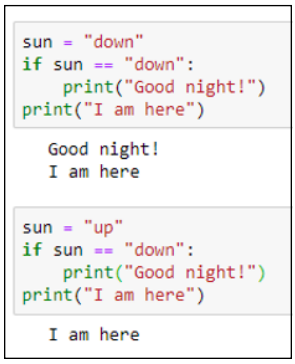


FIGURE 2-3: Result of simple `if` when the condition proves true and then false.

If you're wondering whether it's better to use a single line or multiple lines in your `if` statements, it depends on what you mean by *better*. If you mean *better* in terms of which method executes the fastest, the answer is neither. You won't be able to see a speed difference when executing the code. If by *better* you mean easier for a human programmer to read, most people would prefer the second method, with the code indented under the `if` statement.

Remember, you can indent any number of lines under the `if`, and those indented lines execute only if the condition proves true. If the condition proves false, none of the indented lines are executed. The unindented code under the indented lines is always executed because it's not dependent on the condition. Here is an example with four lines of code that execute only if the condition proves true:

```
total = 100
sales_tax_rate = 0.065
taxable = True
if taxable:
    print(f"Subtotal : ${total:.2f}")
    sales_tax = total * sales_tax_rate
    print(f"Sales Tax: ${sales_tax:.2f}")
    total = total + sales_tax
print(f"Total    : ${total:.2f}")
```

REMEMBER

You must spell `True` and `False` with an initial capital letter and the rest lowercase. If you type it any other way, Python won't recognize it as a Boolean `True` or `False` and your code won't run as expected.

Notice that in the `if` statement we used

```
if taxable:
```

This code is perfectly okay because we made `taxable` a Boolean that can only be `True` or `False`. You may see other people type it as

```
if taxable == True:
```

This line is okay too, and it won't have any negative effect on the code. The `== True` is just unnecessary because, by itself, `taxable` is already either `True` or `False`.

Anyway, as you can see, we start off with a `total` variable, a `sales_tax_rate` variable, and a `taxable` variable. When `taxable` is `True`, all four lines under the `if` are executed, and you end up with the output shown in Figure 2-4.

When `taxable` is set to `False`, all the indented lines are skipped over, and the total shown is the original total without sales tax added, as shown in Figure 2-5.

```
total = 100
sales_tax_rate = 0.065
taxable = True
if taxable:
    print(f"Subtotal : ${total:.2f}")
    sales_tax = total * sales_tax_rate
    print(f"Sales Tax: ${sales_tax:.2f}")
    total = total + sales_tax
print(f"Total    : ${total:.2f}")
```

```
Subtotal : $100.00
Sales Tax: $6.50
Total    : $106.50
```

FIGURE 2-4: When `taxable` is `True`, `sales_tax` is added to the total.

```
total = 100
sales_tax_rate = 0.065
taxable = False
if taxable:
    print(f"Subtotal : ${total:.2f}")
    sales_tax = total * sales_tax_rate
    print(f"Sales Tax: ${sales_tax:.2f}")
    total = total + sales_tax
print(f"Total    : ${total:.2f}")
```

```
Total    : $100.00
```

FIGURE 2-5: When `taxable` is `False`, `sales_tax` is not added into the total.

TECHNICAL STUFF

The curly braces and `.2f` stuff in Figures 2-4 and 2-5 are just for formatting, as we discuss in Book 2, Chapter 1, and have nothing to do with the `if` logic of the code.

Adding else to your if logic

So far you've looked at code examples in which some code is executed if some condition proves true. If the condition proves false, that code is ignored. Sometimes, you may want one chunk of code to execute *if* a condition proves true; *otherwise* (*else*), if it doesn't prove true, you want some other chunk of code to be executed. In that case, you can add an `else:` to your `if`. Any lines of code indented under the `else:` are executed only if the condition did not prove true. Here is the logic and syntax:

```
if condition:
    do indented lines here
     ...
else:
    do indented lines here
     ...
do remaining un-indented lines no matter what
```

Figure 2-6 shows a simple example where we grab the current time from the computer clock using `datetime.now()`. If the hour of that time is less than 12, the program displays `Good morning`. Otherwise, it displays `Good afternoon`. Regardless of the hour, it prints `I hope you are doing well!` So if you write such a program and run it in the morning, you get the appropriate greeting followed by `I hope you are doing well!`, as in Figure 2-6.

```
import datetime as dt
# Get the current date and time
now = dt.datetime.now()
# Make a decision based on hour
if now.hour < 12:
    print("Good morning")
else:
    print("Good afternoon")
print("I hope you are doing well!")
```

```
Good morning
I hope you are doing well!
```

FIGURE 2-6: Print an initial greeting based on the time of day.

Now you may look at that and say "Wow, that's impressive, Einstein. But what if it's 11:00 at night? Do you really want to say "Good afternoon"? Yet another question deserving of a resounding "Hmm." What we need is an `if . . . else` where multiple `else` statements are possible. That's where the `elif` statement, described next, comes into play.

Handling multiple else statements with elif

When `if . . . else` isn't enough to handle all the possibilities, there's `elif` (which, as you may have guessed, is a word made up from `else if`). An `if` statement can include any number of `elif` conditions. You can include or not include a final `else` statement that executes only if the `if` and all the previous `elifs` prove false.

In its simplest form, the syntax for an `if` with `elif` and `else` is

```
if condition:
     do these indented lines of code
      ...
elif condition:
    do these indented lines of code
     ...
do these un-indented lines of code no matter what
```

Given that structure, it's possible that none of the indented code will execute. Take a look at this example:

```
light_color = "green"
if light_color == "green":
    print("Go")
elif light_color == "red":
    print("Stop")
print("This code executes no matter what")
```

Executing that code results in the following:

```
Go
This code executes no matter what
```

If you change the light color to red, like this:

```
light_color = "red"
if light_color == "green":
    print("Go")
elif light_color == "red":
    print("Stop")
print("This code executes no matter what")
```

the result is

```
Stop
This code executes no matter what
```

Suppose you change the light color to anything other than red or green, as follows:

```
light_color = "yellow"
if light_color == "green":
    print("Go")
elif light_color == "red":
    print("Stop")
print("This code executes no matter what")
```

Executing this code produces the following output, because neither `color == "green"` nor `color == "red"` proved true, so none of the indented code was executed:

```
This code executes no matter what
```

You can add an `else` option that happens only if the previous conditions all prove false:

```
light_color = "yellow"
if light_color == "green":
    print("Go")
elif light_color == "red":
    print("Stop")
else:
    print("Proceed with caution")
print("This code executes no matter what")
```

The output is

```
Proceed with caution
This code executes no matter what
```

The fact that the `light_color` is yellow prevents the first two `if` conditions from proving true, so only the `else` code is executed. And that's true for anything that you put into the `light_color` variable, except `"red"` or `"green"`, because the `else` isn't looking for a specific condition. It's just playing an "if all else fails, do this" role in the logic.

Ternary operations

Here is another code example, where we set a variable named `age` to `31`. Then we use `if...elif...else` to make a decision about what to display:

```
age = 31
if age < 21:
    beverage = "milk"
elif age >= 21 and age < 80:
    beverage = "beer"
else:
    beverage = "prune juice"

print("Have a " + beverage)
```

Comments are always optional. But adding comments to the code can make it easier to understand, for future reference:

```
age = 31

if age < 21:
    # If under 21, no alcohol
```

```
    beverage = "milk"

elif age >= 21 and age < 80:
    # Ages 21 - 79, suggest beer
    beverage = "beer"

else:
    # If 80 or older, prune juice might be a good choice.
    beverage = "prune juice"

print("Have a " + beverage)
```


REMEMBER

If you're wondering about the rule for indenting comments, there is no rule. Comments are just notes to yourself; they aren't executable code. So they're never executed like code, no matter what their level indentation.

Repeating a Process with for

Decision-making is a big part of writing all kinds of apps — games, artificial intelligence, robotics . . . whatever. But sometimes you need to count or perform a task over and over. For those times, you can use a *for loop*, which enables you to repeat a line of code, or several lines of code, as many times as you like.

Looping through numbers in a range

If you know how many times you want a loop to repeat, using the following syntax may be easiest:

```
for x in range(y):
   do this
   do this
   ...
un-indented code is executed after the loop
```

Replace *x* with any variable name of your choosing. Replace *y* with any number or range of numbers. If you specify one number, the range will be from 0 to 1 less than the final number. For example, run this code in a Jupyter notebook or `.py` file:

```
for x in range(7):
    print(x)
print("All done")
```

The output is the result of executing `print(x)` once for each pass through the loop, with `x` starting at 0. The final line, which isn't indented, executes after the loop has finished looping. So the output is

```
0
1
2
3
4
5
6
All done
```

You might have expected the loop to count from `1` to `7` instead of `0` to `6`. However, unless you specify otherwise, the loop always starts counting from `0`. If you want to start counting with another number, specify the starting number and the ending number, separated by a comma, inside the parentheses. When you specify two numbers, the first number identifies where the counting starts. The second number is 1 greater than where the loop stops (which is unfortunate for readability but such is life). For example, here is a `for` loop with two numbers in the range:

```
for x in range(1, 10):
    print(x)
print("All done")
```

When you run this code, the counter starts at `1` and, as mentioned, stops 1 short of the last number:

```
1
2
3
4
5
6
7
8
9
All done
```

If you want the loop to count from 1 to 10, the range is `1,11`. This won't make your brain cells any happier, but at least it gets the desired goal of 1 to 10, as shown in Figure 2-7.

```
for x in range(1, 11):
    print(x)
print("All done")
```

```
1
2
3
4
5
6
7
8
9
10
All done
```

FIGURE 2-7: A loop that counts from 1 to 10.

Looping through a string

Using `range()` in a `for` loop is optional. You can replace `range` with a string, and the loop repeats once for each character in the string. The variable `x` (or whatever you name the variable) contains one character from the string with each pass through the loop, going from left to right. The syntax here is

```
for x in string
    do this
    do this
     ...
do this when the loop is done
```

As usual, replace *x* with any variable name you like. The string should be text enclosed in quotation marks, or it should be the name of a variable that contains a string. For example, type this code into a Jupyter notebook or `.py` file:

```
for x in "snorkel":
    print(x)
print("Done")
```

When you run this code, you get the following output. The loop prints one letter from the word *snorkel* with each pass through the loop. When the looping was finished, execution fell to the first un-indented line outside the loop.

```
s
n
o
r
k
e
l
Done
```

The string doesn't have to be a literal string. It can be the name of any variable that contains a string. For example, try this code:

```
my_word = "snorkel"
for x in my_word:
    print(x)
print("Done")
```

The result is the same. The only difference is that we used a variable name rather than a string in the `for` loop. But the code knew that you meant the contents of `my_word` rather than the literal string `my_word`, because `my_word` isn't enclosed in quotation marks.

```
s
n
o
r
k
e
l
Done
```


Looping through a list

In Python, a *list* is basically any group of items, separated by commas, inside square brackets. You can loop through such a list using a `for` loop. In the following example, the list to loop through is specified in brackets on the first line:

```
for x in ["The", "rain", "in", "Spain"]:
    print(x)
print("Done")
```

This kind of loop repeats once for each item in the list. The `x` variable gets its value from one item in the list, going from left to right. So, running the preceding code produces the output you see in Figure 2-8.

You can assign the list to a variable, too, and then use the variable name in the `for` loop rather than the list. Figure 2-9 shows an example where the `seven_dwarves` variable is assigned a list of seven names. Again, note how the list is contained in square brackets. These make Python treat the variable as a list. The *`for`* loop then loops through the list, printing the name of one dwarf (one item in the list) with each pass through the loop. We used the variable name `dwarf` rather than `x`, but that name can be any valid name you like. We could have used `x` or `little_person` or `name_of_fictional_entity` or `goober_wocky` or anything else, as long as the name in the first line matches the name used in the `for` loop.

```
for x in ["The", "rain", "in", "Spain"]:
    print(x)
print("Done")
```

```
The
rain
in
Spain
Done
```

FIGURE 2-8: Looping through a list.

```
seven_dwarves = ["Happy", "Grumpy", "Sleepy", "Bashful", "Sneezy", "Doc", "Dopey"]
for dwarf in seven_dwarves:
    print(dwarf)
print("And Snow White too")
```

```
Happy
Grumpy
Sleepy
Bashful
Sneezy
Doc
Dopey
And Snow White too
```

FIGURE 2-9: Looping through a list.

Bailing out of a loop

Typically, you want a loop to go through an entire list or range of items, but you can also force a loop to stop early if some condition is met. Use the `break` statement inside an `if` statement to force the loop to stop early. The syntax is

```
for x in items:
  if condition:
    [do this ... ]
    break
  do this
```

The square brackets in this example aren't part of the code. They indicate that what is between the brackets is optional. Suppose that someone completed an exam and we want to loop through the answers. But we have a rule that says if an answer is empty, we mark it Incomplete and ignore the rest of the items in the list. In the following, all items are answered (no blanks):

```
answers = ["A", "C", "B", "D"]
for answer in answers:
    if answer == "":
        print("Incomplete")
        break
    print(answer)
print("Loop is done")
```

In the result, all four answers are printed:

```
A
C
B
D
Loop is done
```

Here is the same code, but the third item in the list is blank, as indicated by `""`, which is an empty string:

```
answers = ["A", "C", "", "D"]
for answer in answers:
    if answer == "":
        print("Incomplete")
        break
    print(answer)
print("Loop is done")
```


Here is the output of running that code:

```
A
C
Incomplete
Loop is done
```

So the logic is, as long as some answer is provided, the `if` code is not executed and the loop runs to completion. However, if the loop encounters a blank answer, it prints `Incomplete` and also "breaks" the loop, jumping down to the first statement outside the loop (the final un-indented statement), which prints `Loop is done`.

Looping with continue

You can also use a `continue` statement in a loop, which is kind of the opposite of `break`. Whereas `break` makes code execution jump past the end of the loop and stop looping, `continue` makes it jump back to the top of the loop and continue with the next item (that is, after the item that triggered the `continue`). So here is the same code as the preceding example, but instead of executing a `break` when execution hits a blank answer, it continues with the next item in the list:

```
answers = ["A", "C", "", "D"]
for answer in answers:
    if answer == "":
        print("Incomplete")
```

```
        continue
    print(answer)
print("Loop is done")
```

The output of that code is as follows. It doesn't print the blank answer, it prints `Incomplete`, but then it goes back and continues looping through the rest of the items:

```
A
C
Incomplete
D
Loop is done
```

Nesting loops

It's perfectly okay to *nest* loops — that is, to put loops inside loops. Just make sure you get your indentations right because the indentations determine which loop, if any, a line of code is located within. For example, in Figure 2-10, an outer loop loops through the words `First`, `Second`, and `Third`. With each pass through the loop, it prints a word and then it prints the numbers 1–3 (by looping through a range and adding 1 to each range value).

```
# Outer loop
for outer in ["First", "Second", "Third"]:
    print(outer)
    # Inner loop
    for inner in range(3):
        print(inner + 1)

print("Both loops are done")
#Out of both loops here
```

```
First
1
2
3
Second
1
2
3
Third
1
2
3
Both loops are done
```

FIGURE 2-10: Nested loops.

The loops work because each word in the outer list is followed by the numbers 1–3. The end of the loop is the first un-indented line at the bottom, which doesn't print until the outer loop has completed its process.

Looping with while

As an alternative to looping with `for`, you can loop with `while`. The difference is subtle. With `for`, you generally get a fixed number of loops, one for each item in a range or one for each item in a list. With a `while` loop, the loop keeps going *as long as* (while) some condition is true. Here is the basic syntax:

```
while condition:
   do this ...
   do this ...
do this when the loop is done
```

With `while` loops, you have to make sure that the *`condition`* that makes the loop stop happens eventually. Otherwise, you get an infinite loop that just keeps going and going and going until some error causes it to fail, or until you force it to stop by closing the app, shutting down the computer, or doing some other awkward thing.

Here is an example where the `while` condition runs for a finite number of times due to three things:

- We create a variable named `counter` and give it a starting value (65).
- We say to run the loop *while* `counter` is less than 91.
- Inside the loop, we increase `counter` by 1 (`counter += 1`). Increasing by 1 repeatedly eventually increases `counter` to more than 91, which ends the loop.

The `chr()` function inside the loop displays the ASCII character for the number in `counter`. Going from 65 to 90 is enough to print all the uppercase letters in the alphabet, as in you see in Figure 2-11.

The easy and common mistake to make with this kind of loop is to forget to increment the counter so that it grows with each pass through the loop and eventually makes the `while` condition `False` and stops the loop. In Figure 2-12, we intentionally removed `counter += 1` to cause that error. As you can see, the loop keeps printing `A`. It keeps going until you stop it.

```
counter = 65
while counter < 91:
    print(str(counter) + "=" + chr(counter))
    counter += 1
print("all done")
```

```
65=A
66=B
67=C
68=D
69=E
70=F
71=G
72=H
73=I
74=J
75=K
76=L
77=M
78=N
79=O
80=P
81=Q
82=R
83=S
84=T
85=U
86=V
87=W
88=X
89=Y
90=Z
all done
```

FIGURE 2-11: Looping while `counter` is less than 91.

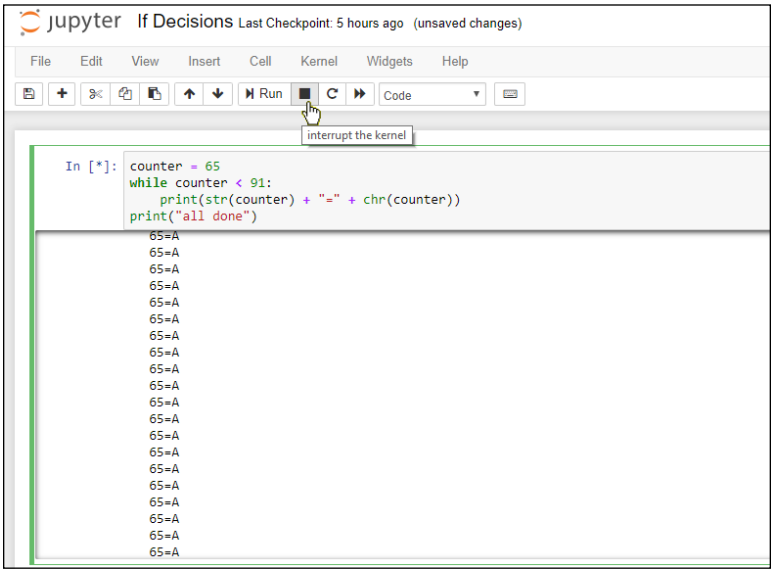


FIGURE 2-12: An infinite `while` loop.

If this happens to you in a Jupyter notebook, don't panic. Just click the square Stop button to the right of Run. (The Stop button displays Interrupt the Kernel, which is nerdspeak for *stop*, when you hover the mouse pointer over it.) All code execution in the notebook will stop. To restart the kernel and get back to square one, click the curved arrow to the right of the Stop button. Then you can fix the error in your code and try again.

Starting while loops over with continue

You can use `if` and `continue` in a `while` loop to skip back to the top of the loop just as you can with `for` loops. Take a look at the code in Figure 2-13 for an example.


```
import random
print("Odd numbers")
counter = 0
while counter < 10:
    # Get a random number
    number = random.randint(1,999)
    if int(number / 2) == number /2:
        # If it's an even number, don't print it.
        continue
    #Otherwise, if it's odd, print it and increment the couter.
    print(number)
    # Increment the loop counter.
    counter += 1
print("Loop is done")
```

```
Odd numbers
697
449
91
567
949
333
591
699
895
837
Loop is done
```

FIGURE 2-13: A `while` loop with `continue`.

A `while` loop keeps going while a variable named `counter` is less than 10. Inside the loop, the variable named `number` is assigned a random number in the range of 1 to 999. Then the following statement checks to see if `number` is even:

```
if int(number / 2) == number / 2:
```

Remember, the `int()` function returns only the whole portion of a number. So let's say the random number that's generated is 5. Dividing this number by 2 gets you 2.5. Then `int(number)` is 2 because the `int()` of a number drops everything after the decimal point. Because 2 doesn't equal 2.5, the code skips over the `continue`, prints that odd number, increments the counter, and keeps going.

If the next random number is, say, 12, well, 12 divided by 2 is 6 and `int(6)` does equal 6 (because neither number has a decimal point). That causes the `continue` to execute, skipping over the `print(number)` statement and the `counter` increment, so it just tries another random number and continues on its merry way. Eventually, it finds 10 odd numbers, at which point the loop stops and the final line of code displays `Loop is done`.

Breaking while loops with break

You can also break a `while` loop using `break`, just as you can with a `for` loop. When you break a `while` loop, you force execution to continue with the first line of code under and outside the loop, thereby stopping the loop but continuing the flow with the rest of the action after the loop.

Another way to think of a `break` is something that allows you to stop a `while` loop before the `while` condition proves false. So it allows you to literally break out of the loop before its time. Truthfully, however, we can't remember a situation where breaking out of a loop before its time was a good solution to a problem, so it's hard to come up with a practical example. In lieu of that, we'll just show you the syntax and provide a generic example. The syntax is

```
while condition1:
    do this
     ...
    if condition2:
        break
do this when the loop is done
```

Basically, two things can stop this loop. Either *condition1* proves false, or *condition2* proves true. Regardless of which of these two things happen, code execution resumes at the first line of code outside the loop, the line that reads `do this code when the loop is done` in the sample code.

Here is an example where the program prints *up to* ten numbers that are not evenly divisible by 5. It may print fewer than that, though, because when it hits a random number that's evenly divisible by 5, it bails out of the loop. So the only thing you

can predict about the example is that it will print between zero and ten numbers that are not evenly divisible by 5. You can't predict how many it will print on any given run, because there's no way to tell if or when it will get a random number evenly divisible by 5 during the ten tries it's allowed:

```
import random
print("Numbers that aren't evenly divisible by 5")
counter = 0
while counter < 10:
    # Get a random number
    number = random.randint(1,999)
    if int(number / 5) == number / 5:
        # If it's evenly divisible by 5, bail out.
        break
    # Otherwise, print it and keep going for a while.
    print(number)
    # Increment the loop counter.
    counter += 1
print("Loop is done")
```

So the first time you run that app, your output may look something like Figure 2-14. The second time you may get something like Figure 2-15. There's just no way to predict the result because the random number is indeed random and not predictable (which is an important concept in many games).

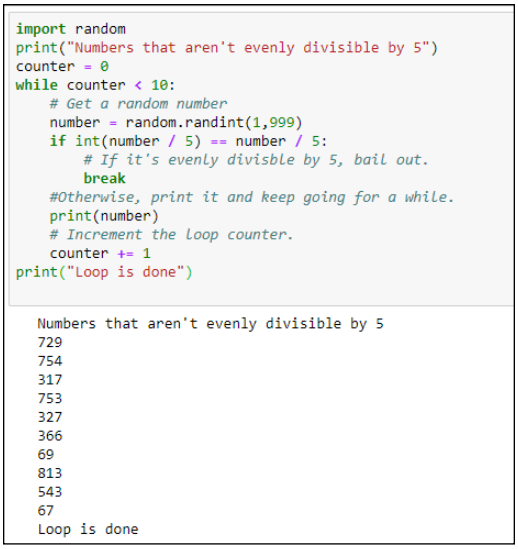


FIGURE 2-14: A `while` loop with `break`.

```
import random
print("Numbers that aren't evenly divisible by 5")
counter = 0
while counter < 10:
    # Get a random number
    number = random.randint(1,999)
    if int(number / 5) == number / 5:
        # If it's evenly divisble by 5, bail out.
        break
    #Otherwise, print it and keep going for a while.
    print(number)
    # Increment the loop counter.
    counter += 1
print("Loop is done")
```

```
Numbers that aren't evenly divisible by 5
866
377
197
Loop is done
```

FIGURE 2-15: The same code as in Figure 2-14 on a second run.

IN THIS CHAPTER

- Defining lists
- Working with lists
- Understanding tuples
- Checking out sets

Chapter 3
Speeding Along with Lists and Tuples

Sometimes in code you work with one item of data at a time, such as a person's name or a unit price or a username. Other times, you work with larger sets of data, such as a list of people's names or a list of products and their prices. These sets of data are often referred to as *lists* or *arrays* in most programming languages.

Python has lots of easy, fast, and efficient ways to deal with all kinds of data collections, as you discover in this chapter. As always, we encourage you to follow along in a Jupyter notebook or `.py` file. The "doing" part helps with the "understanding" part.

Defining and Using Lists

The simplest data collection in Python is a list. We provided examples of these in the preceding chapter. A *list* is any list of data items, separated by commas, inside square brackets. Typically, you assign a name to the list using an = character, just as you would with variables. If the list contains numbers, don't use quotation marks around them. For example, here is a list of test scores:

```
scores = [88, 92, 78, 90, 98, 84]
```

REALLY, REALLY LONG LISTS

All the lists in this chapter are short to make the examples easy and manageable. In real life, however, your lists might contain hundreds or even thousands of items that change frequently. Typing such long lists in the code directly would make the code difficult to work with. Instead, you'd store such lists in external files or external databases, where everything is easier to manage.

All the techniques you learn in this chapter apply to lists stored in external files. The only difference is that you have to write code to pull the data into the list first. But before you start tackling big lists, you need to know all the techniques for working with lists of any size. So stick with this chapter before you move on to managing external data. You'll be glad you did.

If the list contains strings, as always, those strings should be enclosed in single or double quotation marks, as in this example:

```
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]
```

To display the contents of a list on the screen, you can print it just as you would print any regular variable. For example, executing `print(students)` in your code after defining that list displays the following on the screen:

```
['Mark', 'Amber', 'Todd', 'Anita', 'Sandy']
```

This output may not be exactly what you had in mind. But don't worry, Python offers lots of ways to display lists.

Referencing list items by position

Each item in a list has a position number, starting with 0, even though you don't see any numbers. You can refer to any item in the list by its number using the name for the list followed by a number in square brackets. In other words, use this syntax:

```
listname[x]
```

Replace *listname* with the name of the list you're accessing and replace *x* with the position number of the item you want. Remember, the first item is always 0, not 1. For example, in the following first line, we define a list named `students`, and

then print item number 0 from that list. The result, when executing the code, is the name `Mark` displayed:

```
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]
print(students[0])
Mark
```


TECHNICAL STUFF

When reading list items aloud, professionals use the word *sub* before the number. For example, *students[0]* would be spoken as "students sub zero."

The next example shows a list named `scores`. The `print()` function prints the position number of the last score in the list, which is 4 (because the first one is always 0).

```
scores = [88, 92, 78, 90, 84]
print(scores[4])
84
```

If you try to access a list item that doesn't exist, you get an `list index out of range` error. The *index* part is a reference to the number inside the square brackets. For example, Figure 3-1 shows a little experiment in a Jupyter notebook where we created a list of scores and then tried to print `score[5]`. It failed and generated an error because there is no `scores[5]`. There's only `scores[0]`, `scores[1]`, `scores[2]`, `scores[3]`, and `scores[4]` because the counting always starts at 0 with the first one in the list.

```
#Define a list of numbers.
scores = [88, 92, 78, 90, 84]

print(scores[5])

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-240d3b4f5443> in <module>()
      5
      6 #Experiment with the lists
----> 7 print(scores[5])

IndexError: list index out of range
```

FIGURE 3-1: Index out-of-range error because `scores[5]` doesn't exist.

Looping through a list

To access each item in a list, just use a `for` loop with this syntax:

```
for x in list:
```

Replace *x* with a variable name of your choosing. Replace *list* with the name of the list.

TIP

An easy way to make the code readable is to always use a plural for the list name (such as `students`, `scores`). Then you can use the singular name (`student`, `score`) for the variable name. You don't need to use subscript numbers (numbers in square brackets) with this approach either. For example, the following code prints each score in the scores list:

```
for score in scores:
    print(score)
```

Remember to always indent the code that's to be executed in the loop. Figure 3-2 shows a more complete example where you can see the result of running the code in a Jupyter notebook.

```
#Define a list of numbers.
scores = [88, 92, 78, 90, 84]
for score in scores:
    print(score)
print("Done")
```

```
88
92
78
90
84
Done
```

FIGURE 3-2: Looping through a list.

Seeing whether a list contains an item

If you want your code to check the contents of a list to see whether it already contains some item, use `in` *`listname`* in an `if` statement or a variable assignment. For example, the code in Figure 3-3 creates a list of names. Then, two variables store the results of searching the list for the names `Anita` and `Bob`. Printing the contents of each variable displays `True` for the one where the name `Anita` is in the list. The test to see whether `Bob` is in the list proves `False`.

```
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]

# Is Anita in the list?
has_anita = "Anita" in students
print(has_anita)

#Is Bob in the list?
has_bob = "Bob" in students
print(has_bob)
```

```
True
False
```

FIGURE 3-3: Seeing whether an item is in a list.

Getting the length of a list

To determine how many items are in a list, use the `len()` function (short for *length*). Put the name of the list inside the parentheses. For example, type the following code in a Jupyter notebook or at the Python prompt or whatever:

```
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]
print(len(students))
```

Running that code produces this output:

```
5
```

The list has five items, though the index of the last item is always 1 less than the number because Python starts counting at 0. So the last item, `Sandy`, refers to `students[4]` and not `students[5]`.

Adding an item to the end of a list

When you want your code to add an item to the end of a list, use the `.append()` method with the value you want to add inside the parentheses. You can use either a variable name or a literal value inside the quotation marks. For instance, in Figure 3-4, the line `students.append("Goober")` adds the name Goober to the list. The line `students.append(new_student)` adds whatever name is stored in the `new_student` variable to the list. The `.append()` method always adds to the end of the list. So when you print the list, those two new names are at the end.


```
#Create a list of strings (names)
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]

#Add the name Goober to the list
students.append("Goober")

new_student = "Amanda"
#Add whatever name is in new_student to the list.
students.append(new_student)

#Print the entire list
print(students)
```

```
['Mark', 'Amber', 'Todd', 'Anita', 'Sandy', 'Goober', 'Amanda']
```

FIGURE 3-4: Appending two new names to the end of the list.

You can use a test to see whether an item is in a list and then append it only when the item isn't already there. For example, the following code won't add the name Amanda to the list because that name is already in the list:

```
student_name = "Amanda"

#Add student_name but only if not already in the list.
if student_name in students:
    print(student_name + " already in the list")
else:
    students.append(student_name)
    print(student_name + " added to the list")
```

Inserting an item into a list

Whereas the `append()` method adds an item to the end of a list, the `insert()` method adds an item to the list in any position. The syntax for `insert()` is

```
listname.insert(position, item)
```

Replace *`listname`* with the name of the list, *`position`* with the position at which you want to insert the item (for example, `0` to make it the first item, `1` to make it the second item, and so forth). Replace *`item`* with the value, or the name of a variable that contains the value, that you want to put in the list.

For example, the following code makes `Lupe` the first item in the list:

```
# Create a list of strings (names).
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]

student_name = "Lupe"
# Add student name to front of the list.
students.insert(0, student_name)

# Show me the new list.
print(students)
```

If you run the code, `print(students)` will display the list after the new name has been inserted, as follows:

```
['Lupe', 'Mark', 'Amber', 'Todd', 'Anita', 'Sandy']
```

Changing an item in a list

You can change an item in a list using the = assignment operator just like you do with variables. Make sure you include the index number in square brackets to indicate which item you want to change. The syntax is

```
listname[index] = newvalue
```

Replace *listname* with the name of the list; replace *index* with the subscript (index number) of the item you want to change; and replace *newvalue* with whatever you want to put in the list item. For example, take a look at this code:

```
# Create a list of strings (names).
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]
students[3] = "Hobart"
print(students)
```

When you run this code, the output is as follows, because Anita has been changed to Hobart:

```
['Mark', 'Amber', 'Todd', 'Hobart', 'Sandy']
```

Combining lists

If you have two lists that you want to combine into a single list, use the `extend()` function with the following syntax:

```
original_list.extend(additional_items_list)
```

In your code, replace *original_list* with the name of the list to which you'll be adding new list items. Replace *additional_items_list* with the name of the list that contains the items you want to add to the first list. Here is a simple example using lists named `list1` and `list2`. After executing `list1.extend(list2)`, the first list contains the items from both lists, as you can see in the output of the `print()` statement at the end.

```
# Create two lists of Names.
list1 = ["Zara", "Lupe", "Hong", "Alberto", "Jake"]
list2 = ["Huey", "Dewey", "Louie", "Nader", "Bubba"]

# Add list2 names to list1.
list1.extend(list2)

# Print list 1.
```

```
print(list1)

['Zara', 'Lupe', 'Hong', 'Alberto', 'Jake', 'Huey', 'Dewey', 'Louie', 'Nader',
   'Bubba']
```

Easy Parcheesi, no?

Removing list items

Python offers a `remove()` method so you can remove any value from the list. If the item is in the list multiple times, only the first occurrence is removed. For example, the following code displays a list of letters with the letter *C* repeated a few times. Then the code uses `letters.remove("C")` to remove the letter *C* from the list:

```
# Create a list of strings.
letters = ["A", "B", "C", "D", "C", "E", "C"]

# Remove "C" from the list.
letters.remove("C")

# Show me the new list.
print(letters)
```

When you execute this code, you'll see that only the first letter *C* has been removed:

```
['A', 'B', 'D', 'C', 'E', 'C']
```

If you need to remove all of an item, you can use a `while` loop to repeat the `.remove` as long as the item still remains in the list. For example, this code repeats the `.remove` as long as "C" is still in the list:

```
while "C" in letters:
    letters.remove("C")
```

If you want to remove an item based on its position in the list, use `pop()` with an index number rather than `remove()` with a value. If you want to remove the last item from the list, use `pop()` without an index number. For example, the following code creates a list, removes the first item (`0`), and then removes the last item (`pop()` with nothing in the parentheses). Printing the list proves that those two items have been removed:

```
# Create a list of strings.
letters = ["A", "B", "C", "D", "E", "F", "G"]
```

```
# Remove the first item.
letters.pop(0)
# Remove the last item.
letters.pop()

# Show me the new list.
print(letters)
```

Running the code shows that popping the first and last items did, indeed, work:

```
['B', 'C', 'D', 'E', 'F']
```

When you `pop()` an item off the list, you can store a copy of that value in some variable. For example, Figure 3-5 shows the same code as the preceding, but it stores copies of what's been removed in variables named `first_removed` and `last_removed`. At the end it prints the list, and also shows which letters were removed.

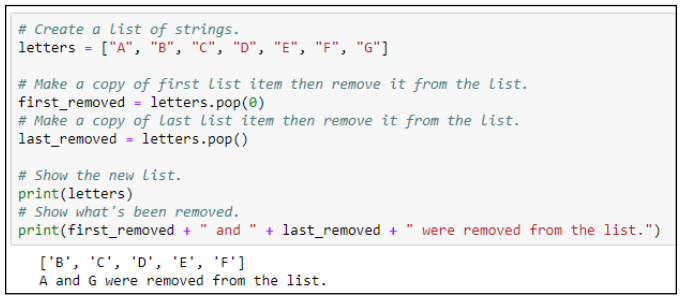


FIGURE 3-5: Removing list items with `pop()`.

Python also offers a `del` (short for *delete*) command that deletes any item from a list based on its index number (position). But again, you have to remember that the first item is `0`. So, let's say you run the following code to delete item number `2` from the list:

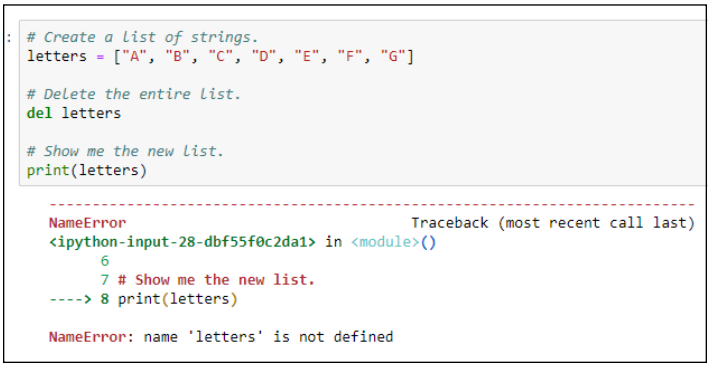
```
# Create a list of strings.
letters = ["A", "B", "C", "D", "E", "F", "G"]

# Remove item sub 2.
del letters[2]

print(letters)
```

Running that code shows the list again, as follows:

```
['A', 'B', 'D', 'E', 'F', 'G']
```

The letter *C* has been deleted, which is the correct item to delete because letters are numbered 0, 1, 2, 3, and so forth.

You can also use `del` to delete an entire list by removing the square brackets and the index number. For example, the code in Figure 3-6 creates a list and then deletes it. Trying to print the list after the deletion causes an error because the list no longer exists when the `print()` statement is executed. Note that unlike `pop`, which returns the item you deleted, `del` just deletes without returning anything.



```
# Create a list of strings.
letters = ["A", "B", "C", "D", "E", "F", "G"]

# Delete the entire list.
del letters

# Show me the new list.
print(letters)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-dbf55f0c2da1> in <module>()
      6
      7 # Show me the new list.
----> 8 print(letters)

NameError: name 'letters' is not defined
```

FIGURE 3-6: Deleting a list and then trying to print it causes an error.

Clearing out a list

If you want to delete the contents of a list but not the list itself, use `.clear()`. The list still exists, but it contains no items. In other words, it's an empty list. The following code shows how you could test this. Running the code displays `[]` at the end, which lets you know the list is empty:

```
# Create a list of strings.
letters = ["A", "B", "C", "D", "E", "F", "G"]

# Clear the list of all entries.
letters.clear()

# Show me the new list.
print(letters)

[]
```

Counting how many times an item appears in a list

You can use the Python `count()` method to count how many times an item appears in a list. As with other list methods, the syntax is simple:

```
listname.count(x)
```

Replace *listname* with the name of your list, and *x* with the value you're looking for (or the name of a variable that contains that value).

The code in Figure 3-7 counts how many times the letter *B* appears in the list, using a literal `B` inside the parentheses of `.count()` like this:

```
grades.count("B")
```

Because `B` is in quotation marks, you know it's a literal, not the name of some variable.

This code also counts the number of *C* grades, but we stored that value in a variable just to show the difference in syntax. Both counts worked, as you can see in the output of the program at the bottom.

We just counted the *F*'s right in the code that displays the message. There are no *F* grades, so `grades.count("F")` returns `0`, as you can see in the output.

In case you're wondering why we're not counting other grades, it's because the app is just an example to illustrate the Python syntax. We're not trying to create an actual product to count all real grades in a classroom.


```
# Create a list of strings.
grades = ["C", "B", "A", "D", "C", "B", "C"]

# Count the B's
b_grades = grades.count("B")

# Use a variable for value to count.
look_for = "C"
c_grades = grades.count(look_for)

print("There are " + str(b_grades) + " B grades in the list.")
print("There are " + str(c_grades) + " " + look_for + " grades in the list.")

#Count Fs too.
print("There are " + str(grades.count("F")) + " F grades in the list.")
```

```
There are 2 B grades in the list.
There are 3 C grades in the list.
There are 0 F grades in the list.
```

FIGURE 3-7: Counting items in a list.

REMEMBER

When trying to combine numbers and strings to form a message, you have to convert the numbers to strings using the `str()` function. Otherwise, you get an error that reads something like `can only concatenate str (not "int") to str`. In that message, `int` is short for *integer* and `str` is short for *string*.

Finding an list item's index

Python offers an `.index()` method that returns a number indicating the position of an item in a list, based on the index number. The syntax is

```
listname.index(x)
```

As always, replace *listname* with the name of the list you want to search. Replace *x* what whatever you're looking for (either a literal or a variable name, as always). Of course, there's no guarantee that the item is in the list or is in the list only once. If the item isn't in the list, an error occurs. If the item is in the list multiple times, the index of only the first matching item is returned.

Figure 3-8 shows an example where the program crashes at the line `f_index = grades.index(look_for)` because there is no *F* in the list.

```
# Create a list of strings.
grades = ["C", "B", "A", "D", "C", "B", "C"]

# Find the index for "B"
b_index = grades.index("B")

#Find the index for F
look_for = "F"
f_index = grades.index(look_for)

# Show the results.
print("The first B is index " + str(b_index))
print("There first "+ look_for + " is at " + str(f_index))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-38-ee447e55d5c6> in <module>()
      7 #Find the index for F
      8 look_for = "F"
----> 9 f_index = grades.index(look_for)
     10
     11 # Show the results.

ValueError: 'F' is not in list
```

FIGURE 3-8: Program fails when trying to find the index of a nonexistent list item.

An easy way to get around this problem is to use an `if` statement to see whether an item is in the list before you try to get its index number. If the item isn't in the list, display a message saying so. Otherwise, get the index number and show it in a message. That code follows:

```
# Create a list of strings.
grades = ["C", "B", "A", "D", "C", "B", "C"]

# Decide what to look for
look_for = "F"
# See if the item is in the list.
if look_for in grades:
    # If it's in the list, get and show the index.
    print(str(look_for) + " is at index " + str(grades.index(look_for)))
else:
    # If not in the list, don't even try for index number.
    print(str(look_for) + " isn't in the list.")
```

Alphabetizing and sorting lists

Python offers a `sort()` method for sorting lists. In its simplest form, it alphabetizes the items in the list (if they're strings). If the list contains numbers, they're sorted smallest to largest. For a simple sort like that, just use `sort()` with empty parentheses:

```
listname.sort()
```

Replace *listname* with the name of your list. Figure 3-9 shows an example using a list of strings and a list of numbers. We created a new list for each simply by assigning each sorted list to a new list name. Then the code prints the contents of each sorted list.

```
# Create a list of strings.
names = ["Zara", "Lupe", "Hong", "Alberto", "Jake", "Tyler"]
# Create a list of numbers
numbers = [14, 0, 56, -4, 99, 56, 11.23]

# Sort the names list.
names.sort()
# Sort the numbers list.
numbers.sort()

# Show the results
print(names)
print(numbers)
```

```
['Alberto', 'Hong', 'Jake', 'Lupe', 'Tyler', 'Zara']
[-4, 0, 11.23, 14, 56, 56, 99]
```

FIGURE 3-9: Sorting strings and numbers.

TIP

If your list contains strings with a mixture of uppercase and lowercase letters, and if the results of the sort don't look right, try replacing `.sort()` with `.sort(key=lambda s: s.lower())` and then running the code again. See Book 2, Chapter 5 if you're curious about the details.

Dates are a little trickier because you can't just type them in as strings, like `"12/31/2020"`. They have to be the `date` data type to sort correctly. This means using the `datetime` module and the `date()` method to define each date. You can add the dates to the list as you would any other list. For example, in the following line, the code creates a list of four dates:

```
dates = [dt.date(2020,12,31), dt.date(2019,1,31), dt.date(2018,2,28),
   dt.date(2020,1,1)]
```

The computer certainly won't mind if you create the list this way. But if you want to make the code more readable to yourself or other developers, you may want to create and append each date, one at a time. Figure 3-10 shows an example where we created an empty list named `datelist`:

```
datelist = []
```

Then we appended one date at a time to the list using the `dt.date(`*year*`,`*month*`,`*day*`)` syntax.

```
]: # Need this modules for the dates.
   import datetime as dt

   # Create a list of dates, empty for starters
   datelist = []
   # Append dates one at time so code is easier to read.
   datelist.append(dt.date(2020,12,31))
   datelist.append(dt.date(2019,1,31))
   datelist.append(dt.date(2018,2,28))
   datelist.append(dt.date(2020,1,1))

   # Sort the dates (earliest to latest) and show formatted.
   datelist.sort()
   for date in datelist:
       print(f"{date:%m/%d/%Y}")

   02/28/2018
   01/31/2019
   01/01/2020
   12/31/2020
```

FIGURE 3-10: Sorting and displaying dates in a nice format.

After the list is created, the code uses `datelist.sort()` to sort the dates into chronological order (earliest to latest). We didn't use `print(datelist)` in that code because that method displays the dates with the data type information included, like this:

```
[datetime.date(2018, 2, 28), datetime.date(2019, 1, 31), datetime.date (2020, 1,
   1), datetime.date(2020, 12, 31)]
```

Not the easiest list to read. So, rather than print the entire list with one `print()` statement, we looped through each date in the list, and printed each one formatted with the f-string `%m/%d/%Y`. This technique displays each date on its own line in *mm/dd/yyyy* format, as you can see at the bottom of Figure 3-10.

If you want to sort items in reverse order, put `reverse=True` inside the `sort()` parentheses (and don't forget to make the first letter of `True` uppercase). Figure 3-11 shows examples of sorting all three lists in descending (reverse) order using `reverse=True`.

```
# Need this module for the dates.
import datetime as dt

# Create a list of strings.
names = ["Zara", "Lupe", "Hong", "Alberto", "Jake", "Tyler"]

# Create a list of numbers.
numbers = [14, 0, 56, -4, 99, 56, 11.23]

# Create a list of dates, empty for starters because code is long.
datelist = []
datelist.append(dt.date(2020,12,31))
datelist.append(dt.date(2019,1,31))
datelist.append(dt.date(2018,2,28))
datelist.append(dt.date(2020,1,1))

# Sort strings in reverse order (Z to A) and show.
names.sort(reverse=True)
print(names)
print() # This just adds a blank line to the output.

#Sort numbers in reverse order (largest to smallest) and show.
numbers.sort(reverse=True)
print(numbers)
print() # This just adds a blank line to the output.

# Sort the dates in reverse order (latest to earliest) and show formatted.
datelist.sort(reverse = True)
for date in datelist:
    print(f"{date:%m/%d/%Y}")
```

```
['Zara', 'Tyler', 'Lupe', 'Jake', 'Hong', 'Alberto']

[99, 56, 56, 14, 11.23, 0, -4]

12/31/2020
01/01/2020
01/31/2019
02/28/2018
```

FIGURE 3-11: Sorting strings, numbers, and dates in reverse order.

Reversing a list

You can also reverse the order of items in a list using the `.reverse` method. This is not the same as sorting in reverse. When you sort in reverse, you still sort: Z–A for strings, largest to smallest for numbers, and latest to earliest for dates. When you reverse a list, you simply reverse the items in the list, no matter their order, without trying to sort them. In the following code, we reverse the order of the names in the list and then print the list.

```
# Create a list of strings.
names = ["Zara", "Lupe", "Hong", "Alberto", "Jake"]
# Reverse the list.
names.reverse()
# Print the list.
print(names)

['Jake', 'Alberto', 'Hong', 'Lupe', 'Zara']
```

Copying a list

If you need to work with a copy of a list so as not to alter the original list, use the `.copy()` method. For example, the following code is similar to the preceding code, except that instead of reversing the order of the original list, we make a copy of the list and reverse that one. Printing the contents of each list shows how the first list is still in the original order whereas the second one is reversed:

```
# Create a list of strings.
names = ["Zara", "Lupe", "Hong", "Alberto", "Jake"]

# Make a copy of the list.
backward_names = names.copy()
# Reverse the copy.
backward_names.reverse()

# Print the list.
print(names)
print(backward_names)

['Zara', 'Lupe', 'Hong', 'Alberto', 'Jake']
['Jake', 'Alberto', 'Hong', 'Lupe', 'Zara']
```

Table 3-1 summarizes the methods you've learned about so far in this chapter. As you will see in upcoming chapters, these methods work with other kinds of *iterables* (a fancy name that means any list or list-like thing that you can go through one at a time).

TABLE 3-1 Methods for Working with Lists

Method	What It Does
`append()`	Adds an item to the end of the list
`clear()`	Removes all items from the list, leaving it empty
`copy()`	Makes a copy of a list
`count()`	Counts how many times an element appears in a list
`extend()`	Appends the items from one list to the end of another list
`index()`	Returns the index number (position) of an element in a list
`insert()`	Inserts an item into the list at a specific position
`pop()`	Removes an element from the list, and provides a copy of that item that you can store in a variable
`remove()`	Removes one item from the list
`reverse()`	Reverses the order of items in the list
`sort()`	Sorts the list in ascending order
`sort(reverse=True)`	Sorts the list in descending order

What's a Tuple and Who Cares?

In addition to lists, Python supports a data structure known as a tuple. Some people pronounce that like "*two*-pull." Some people pronounce it to rhyme with "couple". But it's not spelled *tupple* or *touple*, so our best guess is that it's pronounced "two-pull." (Heck, for all we know, there may not be only one correct way to pronounce it, but that doesn't stop people from arguing about it.)

Anyway, despite the oddball name, a *tuple* is just an immutable list (like that tells you a lot). In other words, a tuple is a list, but you can't change it after it's defined. So why would you want to put immutable, unchangeable data in an app? Consider Amazon. If we could all go into Amazon and change things at will, everything would cost a penny and we'd all have housefuls of Amazon stuff that cost a penny, rather than housefuls of Amazon stuff that cost more than a penny.

The syntax for creating a tuple is the same as the syntax for creating a list, except you don't use square brackets. You have to use parentheses, like this:

```
prices = (29.95, 9.98, 4.95, 79.98, 2.95)
```

Most of the techniques and methods that you learned for using lists back in Table 3-1 *don't* work with tuples because they are used to modify something in a list, and a tuple can't be modified. However, you can get the length of a tuple using `len`, like this:

```
print(len(prices))
```

You can use `.count()` to see how many times an item appears in a tuple. For example:

```
print(prices.count(4.95))
```

You can use `in` to see whether a value exists in a tuple, as in the following sample code:

```
print(4.95 in prices)
```

This returns `True` if the tuple contains `4.95` or `False` if it doesn't.

If an item exists in the tuple, you can get its index number. You'll get an error, though, if the item doesn't exist in the list. You can use `in` first to see whether the item exists before checking for its index number, and then you can return some nonsense value such as `-1` if it doesn't exist, as in this code:

```
look_for = 12345
if look_for in prices:
    position = prices.index(look_for)
else:
    position = -1
print(position)
```

You can loop through the items in a tuple and display them in any format you want by using format strings. For example, this code displays each item with a leading dollar sign and two digits for the pennies:

```
# Loop through and display each item in the tuple.
for price in prices:
    print(f"${price:.2f}")
```

The output from running this code with the sample tuple follows:

```
$29.95
$9.98
$4.95
$79.98
$2.95
```

You can't change the value of an item in a tuple using this kind of syntax:

```
prices[1] = 234.56
```

You'll get an error message that reads `TypeError: 'tuple' object does not support item assignment`. This message is telling you that you can't use the assignment operator, `=`, to change the value of an item in a tuple because a tuple is immutable, meaning its content cannot be changed.

Any method that alters, or even just copies, data in a list causes an error when you try it with a tuple. So the list methods `.append()`, `.clear()`, `.copy()`, `.extend()`, `.insert()`, `.pop()`, `.remove()`, `.reverse()`, and `.sort()` would fail when working with tuples. In short, a tuple makes sense if you want to *show* data to users without giving them any means to *change* any of the information.

Working with Sets

Python also offers *sets* as a means of organizing data. The difference between a set and a list is that the items in a set have no specific order. Even though you may define the set with the items in a certain order, none of the items get index numbers to identify their position.

To define a set, use curly braces where you use square brackets for a list and parentheses for a tuple. For example, here's a set with some numbers in it:

```
sample_set = {1.98, 98.9, 74.95, 2.5, 1, 16.3}
```

Sets are similar to lists and tuples in a few ways. You can use `len()` to determine how many items are in a set. Use `in` to determine whether an item is in a set.

But you can't get an item in a set based on its index number. Nor can you change an item already in the set. You can't change the order of items in a set either. So you can't use `.sort()` to sort the set or `.reverse()` to reverse its order.

You can add a single new item to a set using `.add()`, as in the following example:

```
sample_set.add(11.23)
```

Not that unlike a list, a set never contains more than one instance of a value. So even if you add 11.23 to the set multiple times, the set will still contain only one copy of 11.23.

You can also add multiple items to a set using `.update()`. But the items you're adding should be defined as a list in square brackets, as in the following example:

```
sample_set.update([88, 123.45, 2.98])
```

You can copy a set. However, because the set has no defined order, when you display the copy, its items may not be in the same order as the original set, as shown in this code and its output:

```
# Define a set named sample_set.
sample_set = {1.98, 98.9, 74.95, 2.5, 1, 16.3}
# Show the whole set
print(sample_set)
# Make a copy and show the copy.
ss2 = sample_set.copy()
print(ss2)

{1.98, 98.9, 2.5, 1, 74.95, 16.3}
{16.3, 1.98, 98.9, 2.5, 1, 74.95}
```

Figure 3-12 shows some sample code and its output. The code creates a set named `sample_set` and then uses a variety of `print()` statements to output information. The following line displays the entire set on the screen:

```
print(sample_set)
```

This line displays `6` because the set has six items:

```
print(len(sample_set))
```

And the following line displays `True` because the number `74.95` is in `sample_set`:

```
print(74.95 in sample_set)
```

Comments in the code describe what the rest of the lines do. Note this command inside the loop near the end of the code:

```
print(f"{price:>6.2f}")
```

```
# Define a set named sample_set.
sample_set = {1.98, 98.9, 74.95, 2.5, 1, 16.3}
# Show the whole set
print(sample_set)

# Use len to get the length of a set.
print(len(sample_set))

# Use in to determine if the set contains a value
print(74.95 in sample_set)

# Use add() to add one item to a set.
sample_set.add(11.23)

# Use update() to add a [list] to a set.
sample_set.update([88, 123.45, 2.98])

print("\nSample set after .add() and .update()")
print(sample_set)

# Loop through the set and print each item right-aligned and formatted.
print("\nLoop through set and print each item formatted.")
for price in sample_set:
    print(f"{price:>6.2f}")
```

```
{1.98, 98.9, 2.5, 1, 74.95, 16.3}
6
True

Sample set after .add() and .update()
{1.98, 98.9, 2.5, 1, 2.98, 74.95, 11.23, 16.3, 88, 123.45}

Loop through set and print each item formatted.
  1.98
 98.90
  2.50
  1.00
  2.98
 74.95
 11.23
 16.30
 88.00
123.45
```

FIGURE 3-12: Playing about with Python sets.

Each number is neatly formatted with two digits, because the code uses the f-string `>6.2f`, which right aligns each number with two digits after the decimal point.

Lists and tuples are two of the most commonly used Python data structures. Sets don't seem to get as much play as the other two, but it's good to know about them. A fourth — and widely used — Python data structure is the data dictionary, which you learn about in the next chapter.

IN THIS CHAPTER

- Producing a data dictionary
- Seeing how to loop through a dictionary
- Copying dictionaries
- Deleting items in a dictionary
- Using multi-key dictionaries

Chapter 4

Cruising Massive Data with Dictionaries

Data dictionaries, also called *associative arrays* in some languages, are kind of like lists, which we discuss in Chapter 3 of this minibook. But each item in the list is identified not by its position in the list but by a key. You can define the key, which can be a string or a number. All that matters is that it is unique to each item in the dictionary.

To understand why uniqueness matters, think about phone numbers, email addresses, and Social Security numbers. If two or more people had the same phone number, whenever someone called that number, all those people would get the call. If two or more people had the same email address, all those people would get the same email messages. If two or more people had the same Social Security number, and one of those people was a million dollars behind in their taxes, you better hope you can convince the tax folks that you're not the one who's delinquent, even though your Social Security number is on the past-due bill.

In this chapter, you'll learn all about Python data dictionaries and how to use them in your own applications.

Understanding Data Dictionaries

A *data dictionary* is similar to a list, except that each item in the list has a unique key. The value you associate with a key can be a number, string, list, tuple — just about anything, really. So you can think of a data dictionary as being similar to a table where the first column contains a single item of information unique to that item and the second column, the value, contains information relevant to, and perhaps unique to, that key. In the example in Figure 4-1, the left column contains a key unique to each row. The second column is the value assigned to each key.

Key		Value
"htanaka"	=	"Haru Tanaka"
"ppatel"	=	"Priya Patel"
"bagarcia"	=	"Benjamin Alberto Garcia"
"zmin"	=	"Zhang Min"
"farooqi"	=	"Ayesha Farooqi"
"hajackson"	=	"Hanna Jackson"
"papatel"	=	"Pratyush Aarav Patel"
"hrjackson"	=	"Henry Jackson"

FIGURE 4-1: A data dictionary with keys in the left column and values in the right.

The left column shows an abbreviation for a person's name. Some businesses use names like these when assigning user accounts and email addresses to their employees.

The value corresponding to each key doesn't have to be a string or an integer. It can be a list, or tuple. For example, in the dictionary in Figure 4-2, the value of each key includes a name, a year (perhaps the year of hire or birth year), a number (for example, the number of dependents the person claims for taxes), and a Boolean `True` or `False` value (which may indicate, for example, whether the person has a company cellphone). For now, it doesn't matter what each item of data represents. What matters is that for each key, you have a list (enclosed in square brackets) that contains four pieces of information about that key.

Key		Value
"htanaka"	=	["Haru Tanaka", 2000, 0, True]
"ppatel"	=	["Priya Patel", 2015, 1, False]
"bagarcia"	=	["Benjamin Alberto Garcia", 1999, 2, True]
"zmin"	=	["Zhang Min", 2017, 0, False]
"farooqi"	=	["Ayesha Farooqi", 2001, 1, True]
"hajackson"	=	["Hanna Jackson", 1998, 0, False]
"papatel"	=	["Pratyush Aarav Patel", 2011, 2, True]
"hrjackson"	=	["Henry Jackson", 2016, 0, False]

FIGURE 4-2: A data dictionary with lists as values.

A dictionary may also consist of several different keys, each representing a piece of data. For example, rather than have a row for each item with a unique key, you might make each employee their own little dictionary. Then you can assign a key name to each unit of information. The dictionary for `htanaka`, then, might look like Figure 4-3.

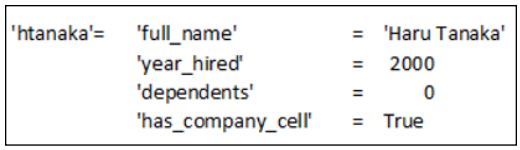


FIGURE 4-3: A data dictionary for one employee.

The dictionary for another employee might have all the same key names, `full_name`, `year_hired`, `dependents`, and `has_company_cell`, but a different value for each of those keys, as shown in Figure 4-4.

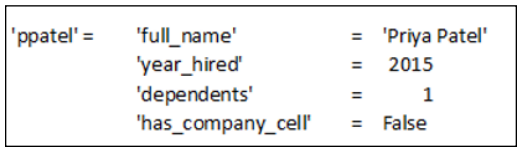


FIGURE 4-4: A data dictionary for another employee.

Each dictionary entry having multiple keys is common in Python, because the language makes it easy to isolate the specific item of data you want using *`object.key`* syntax, like this:

```
ppatel.full_name = 'Priya Patel'
ppatel.year_hired = 2015
ppatel,dependents = 1
ppatel.has_company_cell = True
```

The key name is more descriptive than using an index based on position, as you can see in the following example.

```
ppatel[0] = 'Priya Patel'
ppatel[1] = 2015
ppatel[2] = 1
ppatel[3]=True
```

Creating a Data Dictionary

The code for creating a data dictionary follows this basic syntax:

```
name = {key:value, key:value, key:value, key:value, ...}
```

The *name* is a name you make up and generally describes to whom or what the key–value pairs refer. The *key*:*value* pairs are enclosed in curly braces. The *key* is usually a string enclosed in quotation marks, but you can use integers instead. Each colon (:) separates the key name from the value assigned to it. The *value* is whatever you want to store for that key name, and can be a number, string, list — pretty much anything. The ellipsis (. . .) just means that you can have as many key-value pairs as you want. Just remember to separate *key*:*value* pairs with commas, as shown in the syntax example.

To make the code more readable, developers often place each *key*:*value* pair on a separate line. But the syntax is still the same. The only difference is that a line break follows each comma, as in the following:

```
name = {
    key:value,
    key:value,
    key:value,
    key:value,
     ...
}
```

If you want to try it out, open a Jupyter notebook, a .py file, or a Python prompt, and type the following code. Note that we created a dictionary named people that contains multiple *key*:*value* pairs, each separated by a comma. The keys and values are strings so they're enclosed in quotation marks, and each key is separated from its value with a colon. It's important to keep all that straight; otherwise the code won't work — yes, even one missing or misplaced or mistyped quotation mark, colon, comma, or curly brace can mess up the whole thing:

```
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
}
```

Accessing dictionary data

After you've added the data, you can work with it in a number of ways. Using `print(people)` — that is, a `print()` function with the name of the dictionary in the parentheses — you get a copy of the entire dictionary, as follows:

```
print(people)
{'htanaka': 'Haru Tanaka', 'ppatel': 'Priya Patel', 'bagarcia': 'Benjamin
   Alberto Garcia', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi',
   'hajackson': 'Hanna Jackson', 'papatel': 'Pratyush Aarav Patel', 'hrjackson':
   'Henry Jackson'}
```

Typically this is not what you want. More often, you're looking for one specific item in the dictionary. In that case, use this syntax:

```
dictionaryname[key]
```

where *`dictionaryname`* is the name of the dictionary, and *`key`* is the key value for which you're searching. For example, if you want to know the value of the `zmin` key, you would enter

```
print(people['zmin'])
```

Think of this line as saying "print people sub zmin," where *sub* just means *the specific key*. When you do that, Python returns the value for that one person — the full name for `zmin`, in this example. Figure 4-5 shows that output after running the code in a Jupyter notebook cell.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

print(people['zmin'])
```

```
Zhang Min
```

FIGURE 4-5: Printing the value of the `zmin` key in the `people` dictionary.

Note that in the code, `zmin` is in quotation marks because it's a string. You can use a variable name instead, as long as it contains a string. For example, consider the following two lines of code. The first one creates a variable named `person` and puts the string `'zmin'` into that variable. The next line doesn't require quotation marks because `person` is a variable name:

```
person = 'zmin'
print(people[person])
```

So what do you think would happen if you executed the following code?

```
person = 'hrjackson'
print(people[person])
```

You would see `Henry  Jackson`, the name (value) that goes with the key `'hrjackson'`.

How about if you ran this bit of code?

```
person = 'schmeedledorp'
print(people[person])
```

Figure 4-6 shows what would happen. You get an error because nothing in the `people` dictionary has the key value `'schmeedledorp'`.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Look for a person.
person = 'schmeedledorp'
print(people[person])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-16-3e728d397aa2> in <module>()
     13 # Look for a person.
     14 person = 'schmeedledorp'
---> 15 print(people[person])

KeyError: 'schmeedledorp'
```

FIGURE 4-6: Python's way of saying there is no *schmeedledorp*.

Getting the length of a dictionary

The number of items in a dictionary is considered its *length.* As with lists, you can use the `len()` statement to determine a dictionary's length. The syntax is

```
len(dictionaryname)
```

As always, replace *`dictionaryname`* with the name of the dictionary you're checking. For example, the following code creates a dictionary, and then stores its length in the `howmany` variable:

```
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }
# Count the number of key:value pairs and put in a variable.
howmany = len(people)
# Show how many.
print(howmany)
```

When executed, the `print` statement shows `8`, the value of the `hominy` variable, as determined by the number of key-value pairs in the dictionary.

TIP

As you may have guessed, an empty dictionary that contains no key-value pairs has a length of `0`.

Seeing whether a key exists in a dictionary

You can use the `in` keyword to see whether a key exists. If the key exists, `in` returns `True`. If the key doesn't exist, `in` returns `False`. Figure 4-7 shows a simple example with two `print()` statements. The first one checks to see whether `hajackson` exists in the dictionary. The second checks to see whether `schmeedledorp` exists in the dictionary.

As you can see, the first `print()` statement shows `True` because `hajackson` is in the dictionary. The second one returns `False` because `schmeedledorp` isn't in the dictionary.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Is there an hajackson in the people dictionary?
print('hajackson' in people)

# Is there an schmeedledorp in the people dictionary?
print('schmeedledorp' in people)
```

```
True
False
```

FIGURE 4-7: Seeing if a key exists in a dictionary.

Getting dictionary data with get()

Having the program crash and burn when you look for something that isn't in the dictionary is a little harsh. A more elegant way to handle that situation is to use the `.get()` method of a data dictionary. The syntax is

```
dictionaryname.get(key)
```

Replace *dictionaryname* with the name of the dictionary you're searching. Replace *key* with the thing you're looking for. Note that `get()` uses parentheses, not square brackets. If you look for something that *is* in the dictionary, such as the following, you'd get the same result as you would using square brackets:

```
# Look for a person.
person = 'bagarcia'
print(people.get(person))
```

What makes `.get()` different is what happens when you search for a non-existent name. You don't get an error, and the program doesn't crash and burn. Instead, `get()` gracefully returns the word `None` to let you know that no person named `schmeedledorp` is in the `people` dictionary, as you can see in Figure 4-8.

You can pass two values to `get()`; the second value is what you want `get` to return if it fails to find what you're looking for. For instance, in the following line of code, we search for `schmeedledorp` again. But this time, if the code doesn't find that person, it displays not `None` but the more pompous message `Unbeknownst to this dictionary`:

```
print(people.get('schmeedledorp', 'Unbeknownst to this dictionary'))
```

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Look for a person.
person = 'schmeedledorp'
print(people.get(person))
```

```
None
```

FIGURE 4-8: Python's nicer way of saying there is no *schmeedledorp*.

Changing the value of a key

Dictionaries are *mutable*, which means you can change the contents of the dictionary from code (not that you can make the dictionary shut up). The syntax is simply

```
dictionaryname[key] = newvalue
```

Replace *dictionaryname* with the name of the dictionary, *key* with the key that identifies the item, and *newvalue* with whatever you want the new value to be.

For example, supposed Hanna Jackson gets married and changes her name to Hanna Jackson-Smith. You want to keep the same key but change the value. The line that reads `people['hajackson'] = "Hanna Jackson-Smith"` makes the change. The `print()` statement below that line shows the value of `hajackson` after executing that line of code. As you can see in Figure 4-9, the name has indeed been changed to `Hanna Jackson-Smith`.

```
# Print hajackson's current value.
print(people['hajackson'])

# Change the value of the hajackson key.
people['hajackson'] = "Hanna Jackson-Smith"

#Print the hajackson key to verify that the value has changed.
print(people['hajackson'])
```

```
Hanna Jackson
Hanna Jackson-Smith
```

FIGURE 4-9: Changing the value associated with a key in a dictionary.

In real life, the data in a dictionary would probably be stored also in some kind of external file so that it's permanent. Additional code would be required to save the dictionary changes to that external file. But you need to learn these basics before you get into all of that, so let's just forge ahead with dictionaries for now.

Adding or changing dictionary data

You can use the dictionary `update()` method to add a new item to a dictionary or to change the value of a current key. The syntax is

```
dictionaryname.update(key, value)
```

Replace *dictionaryname* with the name of the dictionary. Replace *key* with the key of the item you want to add or change. If the key you specify doesn't exist in the dictionary, it will be added as a new item with the *value* you specify. If the `key` you specify does exist, nothing will be added. The value of the key will be changed to whatever you specify as the *value*.

For example, consider the following Python code that creates a data dictionary named `people` and put two peoples' names into it:

```
# Make a data dictionary named people.
people = {
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Change the value of the hrjackson key.
people.update({'hrjackson' : 'Henrietta Jackson'})
print(people)

# Update the dictionary with a new key:value pair.
people.update({'wwiggins' : 'Wanda Wiggins'})
```

The first update line changes the value for `hrjackson` from `Henry Jackson` to `Henrietta Jackson` because the `hrjackson` key already exists in the data dictionary:

```
people.update({'hrjackson' : 'Henrietta Jackson'})
```

The second `update()` reads as follows:

```
people.update({'wwiggins' : 'Wanda Wiggins'})
```

There is no `wwiggins` key in the dictionary, so `update()` can't change the name for `wwiggins`. Instead, the line adds a new key-value pair to the dictionary with `wwigins` as the `key` and `Wanda Wiggins` as the `value`.

The code doesn't specify whether to change or add the value because the decision is made automatically. Each key in a dictionary must be unique; you can't have two or more rows with the same key. So when you do an `update()`, the code first checks to see whether the key exists. If it does, only the value of that key is modified; nothing new is added. If the key doesn't exist in the dictionary, there is nothing to modify so the new key-value is added to the dictionary. That process is automatic, and the decision about which action to perform is simple:

- If the key already exists in the dictionary, its value is updated because no two items in a dictionary are allowed to have the same key.
- If the key does *not* already exist, the key-value pair is added because nothing in the dictionary already has that key, so the only choice is to add it.

After running the code, the dictionary contains three items, `paptel`, `hrjackson` (with the new name), and `wwiggins`. Adding the following lines to the end of that code displays everything in the dictionary:

```
# Show what's in the data dictionary now.
for person in people.keys():
    print(person + " = " + people[person])
```

If you add that code and run it again, you get the following output, which shows the complete contents of the data dictionary at the end of that program:

```
papatel = Pratyush Aarav Patel
hrjackson = Henrietta Jackson
wwiggins = Wanda Wiggins
```

As you may have guessed, you can loop through a dictionary in much the same way you loop through lists, tuples, and sets. But you can do some extra things with dictionaries, so let's take a look at those next.

Looping through a Dictionary

You can loop through each item in a dictionary in much the same way you can loop through lists and tuples, but you have some extra options. If you just specify the dictionary name in the `for` loop, you get all the keys, as follows:

```
for person in people:
    print(person)

htanaka
ppatel
bagarcia
zmin
afarooqi
hajackson
papatel
hrjackson
```

If you want to see the value of each item, keep the `for` loop the same, but print *`dictionaryname`*`[`*`key`*`]` where *`dictionaryname`* is the name of the dictionary (`people` in our example) and *`key`* is whatever name you use right after the `for` in the loop (`person`, in the following example).

```
for person in people:
    print(people[person])
```

Running this code against the sample `people` dictionary lists all the names, as follows:

```
Haru Tanaka
Priya Patel
Benjamin Alberto Garcia
Zhang Min
Ayesha Farooqi
Hanna Jackson
Pratyush Aarav Patel
Henry Jackson
```

You can also get all the names by using a slightly different syntax in the `for` loop: Add `.values()` to the dictionary name, as in the following. Then you can just print the variable name (`person`) inside the loop. The output would be the full name of each person, as in the previous loop example.

```
for person in people.values():
    print(person)
```

Lastly, you can loop through the keys and values at the same time by using `.items()` after the dictionary name in the `for` loop. But you will need two variables after the `for` as well, one to reference the key and the other to reference the value. If you want the code to display both variables as it's looping through the dictionary, you'll need to use those names inside the parentheses of the `print`.

For example, the loop in Figure 4-10 uses two variable names, `key` and `value` (although they could be `x` and `y` or anything else) to loop through `people.items()`. The `print` statement displays both the `key` and the `value` with each pass through the loop. The `print()` also has an equal sign (enclosed in quotation marks) to separate the key from the value. As you can see in the output, you get a list of all the keys followed by an equal sign and the value assigned to that key.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Loop through .items to get the key and the value.
for key, value in people.items():
    # Show the key and value with = in between.
    print(key, "=", value)
```

```
htanaka = Haru Tanaka
ppatel = Priya Patel
bagarcia = Benjamin Alberto Garcia
zmin = Zhang Min
afarooqi = Ayesha Farooqi
hajackson = Hanna Jackson
papatel = Pratyush Aarav Patel
hrjackson = Henry Jackson
```

FIGURE 4-10: Looping through a dictionary with `items()` and two variable names.

Data Dictionary Methods

If you've been diligently following along chapter to chapter, you may have noticed that some of the methods for data dictionaries look similar to those for lists, tuples, and sets. So maybe now would be a good time to list, in Table 4-1, all the methods that dictionaries offer. You've already seen some put to use in this chapter. We get to the others a little later.

TABLE 4-1 Data Dictionary Methods

Method	What It Does
`clear()`	Empties the dictionary by remove all keys and values.
`copy()`	Returns a copy of the dictionary.
`fromkeys()`	Returns a new copy of the dictionary but with only specified keys and values.
`get()`	Returns the value of the specified key, or None if it doesn't exist.
`items()`	Returns a list of items as a tuple for each key-value pair.
`keys()`	Returns a list of all the keys in a dictionary.
`pop()`	Removes the item specified by the key from the dictionary, and returns its value.
`popitem()`	Removes the last key-value pair.
`setdefault()`	Returns the value of the specified key. If the key doesn't exist, inserts the key with the specified value.
`update()`	Updates the value of an existing key, or adds a new key-value pair if the specified key isn't already in the dictionary.
`values()`	Returns a list of all the values in the dictionary.

Copying a Dictionary

If you need to make a copy of a data dictionary to work with, use this syntax:

```
newdictionaryname = dictionaryname.copy()
```

Replace *newdictionaryname* with whatever you want to name the new dictionary. Replace *dictionaryname* with the name of the existing dictionary that you want to copy.

Figure 4-11 shows a simple example in which we created a dictionary named `people`, and then created a dictionary named `peeps2` as a copy of the `people` dictionary. Printing the contents of each dictionary shows that they're identical.

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }

# Make a copy of the people dictionary and put it in peeps 2.
peeps2 = people.copy()

#Show what's in both dictionaries
print(people)
print(peeps2)
```

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
```

FIGURE 4-11: Copying a dictionary.

Deleting Dictionary Items

You can remove data from data dictionaries in several ways. The `del` keyword (short for *delete*) can remove any item based on its key. The syntax is as follows:

```
del dictionaryname[key]
```

For example, the following code creates a dictionary named `people`. Then it uses `del people["zmin"]` to remove the item that has `zmin` as its key:

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }

# Show original people dictionary.
print(people)

# Remove zmin from the dictionary.
del people["zmin"]

# Show what's in people now.
print(people)
```

Printing the contents of the dictionary shows that `zmin` is no longer in that dictionary:

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
{'htanaka': 'Haru Tanaka', 'afarooqi': 'Ayesha Farooqi'}
```

If you forget to include a specific key with the `del` keyword and specify only the dictionary name, the entire dictionary is deleted, even its name. For example,

suppose you executed `del people` instead of using `del people["zmin"]` in the preceding code. The output of the second `print(people)` would be an error, as in the following, because after the `people` dictionary is deleted it no longer exists and its content can't be displayed:

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
------------------------------------------------------------
NameError Traceback (most recent call last)
<ipython-input-32-24401f5e8cf0> in <module>()
13
14 # Show what's in people now.
---> 15 print(people)
NameError: name 'people' is not defined
```

To remove all key-value pairs from a dictionary without deleting the entire dictionary, use the `clear` method with this syntax:

```
dictionaryname.clear()
```

The following code creates a dictionary named `people`, puts some key-value pairs in it, and then prints the dictionary so you can see its content. Then, `people.clear()` empties all the data:

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }

# Show original people dictionary.
print(people)

# Remove all data from the dictionary.
people.clear()

#Show what's in people now.
print(people)
```

The output of running this code shows that the `people` data dictionary initially contains three property:value pairs. After using `people.clear()` to wipe the `people` dictionary clear, printing it displays `{}`, which is Python's way of telling you that the dictionary is empty.

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
{}
```

The pop() method offers another way to remove data from a data dictionary. The pop() method actually does two things:

- If you store the results of the pop() method in a variable, that variable gets the value of the popped key.
- Regardless of whether you store the result of the pop() method in a variable, the specified key is removed from the dictionary.

Figure 4-12 shows an example where you first see the entire dictionary in the output. Then `adios = people.pop("zmin")` is executed, putting the value of the `zmin` key in a variable named `adios`. We then print the `adios` variable so we can see that it contains `Zhang Min`, the value of the `zmin` key. Printing the entire `people` dictionary again proves that `zmin` has been removed from the dictionary.

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }
# Show original people dictionary.
print(people)

# Pop zmin from the dictionary, store its value in adios variable.
adios = people.pop("zmin")

# Print the contents of adios and people.
print(adios)
print(people)
```

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
Zhang Min
{'htanaka': 'Haru Tanaka', 'afarooqi': 'Ayesha Farooqi'}
```

FIGURE 4-12: Popping an item from a dictionary.

Data dictionaries offer a variation on pop() that uses this syntax:

```
dictionaryname = popitem()
```

This syntax is tricky because in some earlier versions of Python it would remove an item at random. That's weird unless you're writing a game or something and want to remove things at random. But as of Python version 3.7 (the version used in this book), popitem() always removes the last key-value pair.

If you store the results of popitem in a variable, you *don't* get that item's value, which is different from the way pop() works. Instead, you get both the key and its value. The dictionary no longer contains that key-value pair. So, in other words, if

you replace `adios = people.pop("zmin")` in Figure 4-12 with `adios = people.popitem()`, the output will be as follows:

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}

('afarooqi', 'Ayesha Farooqi')

{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min'}
```

Having Fun with Multi-Key Dictionaries

So far you've worked with a dictionary that has one value (a person's name) for each key (an abbreviation of that person's name). But it's not unusual for a dictionary to have multiple key-value pairs for one item of data.

For example, suppose that just knowing the person's full name isn't enough. You want to also know the year the person was hired, his or her date of birth, and whether or not that employee has been issued a company laptop. The dictionary for any one person might look like this:

```
employee = {
    'name': 'Haru Tanaka',
    'year_hired': 2005,
    'dob': '11/23/1987',
    'has_laptop': False
}
```

Or suppose you need a dictionary of products that you sell. For each product, you want to know its name, its unit price, whether or not it's taxable, and how many you currently have in stock. The dictionary might look something like this (for one product):

```
product = {
    'name': 'Ray-Ban Wayfarer Sunglasses',
    'unit_price': 112.99,
    'taxable': True,
    'in_stock'=: 10
}
```

Note that in each example, the key name is in quotation marks. We used single quotes in the sample code, but you can use either single or double quotes. We even enclosed the date in `dob` (date of birth) in quotation marks. If you don't, it may be treated as a set of numbers, as in "11 divided by 23 divided by 1987" which isn't useful information. Booleans are either `True` or `False` (initial caps) with no

quotation marks. Integers (2005, 10) and floats (112.99) are not enclosed in quotation marks either.

The value for a property can be a list, tuple, or set; it doesn't have to be a single value. For example, for the sunglasses product, maybe you offer two models, black and tortoise. You could add a `colors` or `model` key and list the items as a comma-separated list in square brackets like this:

```
product = {
    'name': 'Ray-Ban Wayfarer Sunglasses',
    'unit_price': 112.99,
    'taxable': True,
    'in_stock': 10,
    'models': ['Black', 'Tortoise']
}
```

Next let's look at how you might display the dictionary data. You can use the simple *`dictionaryname`*`[`*`key`*`]` syntax to print just the value of each key. For example, using that last product example, the output of this code:

```
print(product['name'])
print(product['unit_price'])
print(product['taxable'])
print(product['in_stock'])
print(product['models'])
```

would be:

```
Ray-Ban Wayfarer Sunglasses
112.99
True
10
['Black', 'Tortoise']
```

You could get fancier by adding descriptive text to each `print` statement, followed by a comma and the code. You could also loop through the list to print each model on a separate line. And you can use an f-string to format the data. For example, here is a variation on the previous `print()` statements:

```
product = {
    'name' : 'Ray-Ban Wayfarer Sunglasses',
    'unit_price' : 112.99,
    'taxable' : True,
    'in_stock' : 10,
    'models' : ['Black', 'Tortoise']
}
```

```
print('Name:    ', product['name'])
print('Price:   ', f"${product['unit_price']:.2f}")
print('Taxable: ', product['taxable'])
print('In Stock:', product['in_stock'])
print('Models:')
for model in product['models']:
    print(" " * 10 + model)
```

Here is the output of that code:

```
Name:     Ray-Ban Wayfarer Sunglasses
Price:    $112.99
Taxable:  True
In Stock: 10
Models:
          Black
          Tortoise
```

WARNING

The `" " * 10` on the last line of code means *print a space (" ") ten times.* In other words, indent ten spaces. If you don't put exactly one space between those quotation marks, you won't get 10 spaces. You'll get 10 of whatever is between the quotation marks, which also means you'll get nothing if you don't put anything between the quotation marks.

Using the mysterious fromkeys and setdefault methods

Data dictionaries in Python offer two methods, named `fromkeys()` and `setdefault()`, which are the cause of much head-scratching among Python learners — and rightly so because it's not easy to find practical applications for their use. But we'll take a shot at it and at least show you what to expect if you ever use these methods in your code.

The `fromkeys()` method uses this syntax:

```
newdictionaryname = dict.fromkeys(iterable[,value])
```

Replace *newdictionary* with whatever you want to name the new dictionary. It doesn't have to be a generic name like *product*. It can be something that uniquely identifies the product, such as a UPC (Universal Product Code) or SKU (stock-keeping unit) specific to your business.

Replace the *iterable* part with any iterable — meaning, something the code can loop through; a simple list will do. The *value* part is optional. If omitted, each key

in the dictionary gets a value of `None`, which is simply Python's way of saying *no value has been assigned to this key in this dictionary yet*.

In the following example, we created a dictionary named `DWC001` (the SKU for a product in our inventory). We gave it a list of key names, enclosed in square brackets and separated by commas, which makes it a properly defined list for Python. We provided nothing for *`value`*. The code then prints the new dictionary. As you can see, the last line of code prints the dictionary, which contains the specified key names with each key having a value of `None`.

```
DWC001 = dict.fromkeys(['name', 'unit_price', 'taxable', 'in_stock', 'models'])
print(DWC001)
{'name': None, 'unit_price': None, 'taxable': None, 'in_stock': None, 'models':
   None}
```

Now, suppose that you don't want to type all those key names. You just want to use the same keys you're using in other dictionaries. In that case, you can use *`dictionary`*`.keys()` for your iterable list of key names, as long as *`dictionary`* refers to another dictionary that exists in the program.

For example, in the following code, we created a dictionary named `product` that has some key names and nothing specific for the values. Then we used `DWC001 = dict.fromkeys(product.keys())` to create a dictionary with the name `DWC001` that has the same keys as the generic `product` dictionary. We didn't specify any values in the `dict.fromkeys(product.keys())` line, so each of those keys in the new dictionary will have values set to `None`.

```
# Create a generic dictionary for products named product.
product = {
    'name': '',
    'unit_price': 0,
    'taxable': True,
    'in_stock': 0,
    'models': []
}
# Create a dictionary named DWC001 that has the same keys as product.
DWC001 = dict.fromkeys(product.keys())

# Show what's in the new dictionary.
print(DWC001)
```

The final `print()` statement shows what's in the new dictionary. You can see it has all the same keys as the `product` dictionary, with each value set to `None`.

```
{'name': None, 'unit_price': None, 'taxable': None, 'in_stock': None, 'models':
   None}
```

The `.setdefault()` value lets you add a new key to a dictionary, with a predefined value. But `.setdefault()` only adds a new key and value; it doesn't alter the value for an existing key, even if that key's value is `None`. So it could come in handy after the fact if you defined other dictionaries and then later wanted to add another `property:value` pair only to dictionaries that don't already have that property.

Figure 4-13 shows an example in which we created the `DWC001` dictionary using the same keys as the `product` dictionary. After the dictionary is created, `setdefault('taxable', True)` adds a key named `taxable` and sets its value to `True` — but only if that dictionary doesn't already have a key named `taxable`. It also adds a key named `reorder_point` and sets its value to `10` but, again, only if that key doesn't already exist.

```
# Create a generic dictionary for products name product.
product = {
    'name': '',
    'unit_price': 0,
    'taxable': True,
    'in_stock': 0,
    'models': []
}
# Create a dictionary for product SKU # DWC001
DWC001 = dict.fromkeys(product.keys())
DWC001.setdefault('taxable',True)
DWC001.setdefault('models',[])
DWC001.setdefault('reorder_point',100)

# Show what's in the new dictionary.
print("Dictionary after fromkeys() and setdefault()")
print(DWC001)

# Change the taxable field from None to True
print("\nDictionary after fromkeys() and setdefault()")
DWC001['taxable']=True

#Print the dictionary after changing taxable to True
print(DWC001)
```

```
Dictionary after fromkeys() and setdefault()
{'name': None, 'unit_price': None, 'taxable': None, 'in_stock': None, 'models': None, 'reorder_point': 100}

Dictionary after fromkeys() and setdefault()
{'name': None, 'unit_price': None, 'taxable': True, 'in_stock': None, 'models': None, 'reorder_point': 100}
```

FIGURE 4-13: Experimenting with `fromkeys` and `setdefault`.

As you can see in the output from the code, after the `fromkeys` and `setdefault` operations, the new dictionary has the same keys as the `product` dictionary plus a new key-value pair, `reorder_point: 10`, which was added by the second `setdefault`. The `taxable` key in that output, though, is still `None`, because `setdefault` won't change the value of an existing key. It adds a new key with the default value to a dictionary only if it doesn't already have that key.

So what if you really did want to set the default of `taxable` to `True`, rather than `None`? The simple solution would be to use the standard syntax, *dictionaryname* [*key*] = *newvalue* to change the value of the extant `taxable` key from `None` to `True`. The second output in Figure 4-13 proves that changing the value of the key in that manner did work.

Nesting dictionaries

By now it may have occurred to you that any given program you write may require several dictionaries, each with a unique name. But if you just define a bunch of dictionaries with names, how could you loop through the whole kit-and-caboodle without specifically accessing each dictionary by name? The answer is, make each dictionary a key-value pair in some containing dictionary, where the key is the unique identifier for each dictionary (for example, a UPC or SKU for each product). The value for each key would then be a dictionary of all the key-value pairs for that dictionary. So the syntax would be:

```
containingdictionaryname = {
    key: {dictionary},
    key: {dictionary},
    key: {dictionary},
    ...
}
```

That's just the syntax for the dictionary of dictionaries. You have to replace all the italicized placeholder names as follows:

- *containingdictionaryname*: This is the name assigned to the dictionary as a whole. It can be any name you like but should describe what the dictionary contains.
- *key*: Each key value must be unique, such as the UPC or SKU for a product, or the username for a person, or even just some sequential number, as long as it's never repeated.
- {*dictionary*} Enclose all the key-value pairs for that one dictionary item in curly braces, and follow that with a comma if another dictionary follows.

Figure 4-14 shows an example in which we have a dictionary named `products` (plural, because it contains many products). This dictionary in turn contains four individual products. Each product has a unique key: `RB0011`, `DWC0317`, and so forth, which are in-house SKU numbers that the business uses to manage its own inventory. Each of those four products in turn has `name`, `price`, and `models` keys.

The complex syntax with all the curly braces, commas, and colons makes it hard to see what's going on (and hard to type). Outside Python, in a text file, a spreadsheet, a database, or wherever you're putting the data, the same data could be stored as a simple table named `Products` with the key names as column headings, like the one in Table 4-2.

```
# Create a generic products dictionary to contain multiple product dictionaries.
products = {
    'RB00111': {'name': 'Ray-Ban Sunglasses', 'price': 112.98, 'models': ['black', 'tortoise']},
    'DWC0317': {'name': 'Drone with Camera', 'price': 72.95, 'models': ['white', 'black']},
    'MTS0540': {'name': 'T-Shirt', 'price': 2.95, 'models': ['small', 'medium', 'large']},
    'ECD2989': {'name': 'Echo Dot', 'price': 29.99, 'models': []}
}
```

FIGURE 4-14: Multiple product dictionaries contained in a larger `products` dictionary.

TABLE 4-2 A Table of Products

ID (key)	Name	Price	Models
RB00111	Ray-Ban Sunglasses	112.98	black, tortoise
DWC0317	Drone with Camera	72.95	white, black
MTS0540	T-Shirt	2.95	small, medium, large
ECD2989	Echo Dot	29.99	

Using a combination of f-strings and some loops, you could get Python to display that data from the data dictionaries in a neat, tabular format. Figure 4-15 shows an example of such code in a Jupyter notebook, with the output from that code right below it.

```
# Create a generic products dictionary to contain multiple product dictionaries.
products = {
    'RB00111': {'name': 'Ray-Ban Sunglasses', 'price': 112.98, 'models': ['black', 'tortoise']},
    'DWC0317': {'name': 'Drone with Camera', 'price': 72.95, 'models': ['white', 'black']},
    'MTS0540': {'name': 'T-Shirt', 'price': 2.95, 'models': ['small', 'medium', 'large']},
    'ECD2989': {'name': 'Echo Dot', 'price': 29.99, 'models': []}
}
# This header shows above the output.
print(f"{'ID':<6} {' Name':<17} {'Price':>8}  {' Models'}")
print('-' * 60) # Prints 60 hyphens.
# Loop through each dictionary in the products dictionary
for oneproduct in products.keys():
    # Get the id of one product.
    id = oneproduct
    # Get the name of one product.
    name = products[oneproduct]['name']
    # Get the unit price of one product and format with $
    unit_price = '$' + f"{products[oneproduct]['price']:,.2f}"
    # Create and empty string variable named models
    models = ''
    # Loop through the models list and tack onto models
    # one item from the list followed by a comma and a space.
    for m in products[oneproduct]['models']:
        models += m + ', '
    # If the models variable is more than two characters in length,
    # Peel off the last two characters (last comma and space).
    if len(models) > 2:
        models = models[:-2]
    else:
        # Otherwise, if no models, show <none>.
        models = "<none>"
    # Print all the variables with a neat f-string.
    print(f"{id:<6} {name:<17} {unit_price:>8} {models}")
# Any unidented code down here executed after the loop completes.
```

```
ID     Name              Price   Models
------------------------------------------------------------
RB00111 Ray-Ban Sunglasses  $112.98 black, tortoise
DWC0317 Drone with Camera   $72.95 white, black
MTS0540 T-Shirt              $2.95 small, medium, large
ECD2989 Echo Dot            $29.99 <none>
```

FIGURE 4-15: Printing data dictionaries formatted into rows and columns.

IN THIS CHAPTER

- Creating your own function
- Including a comment in a function
- Seeing how to pass information to a function
- Returning values from a function
- Understanding anonymous functions

Chapter 5

Wrangling Bigger Chunks of Code

In this chapter, you learn how to better manage larger code projects by creating your own functions. Functions provide a way to compartmentalize your code into small tasks that can be called from multiple places in an app. For example, if something you need to access throughout the app requires a dozen lines of code, chances are you don't want to repeat that code over and over every time you need it. Doing so just makes the code larger than it needs to be. Also, if you want to change something, or if you have to fix an error in that code, you don't want to have to do it repeatedly in a bunch of different places. If all that code were contained in a function, you would have to change or fix it in only one location.

To access the task that the function performs, you *call* the function from your code, just like you call a built-in function such as `print`. In other words, you just type the name into your code. You can make up your own function names, too. So, think of functions as a way to personalize the Python language so that its commands fit what you need in your application.

Creating a Function

Creating a function is easy. Follow along in a Jupyter notebook cell or `.py` file if you want to get some hands-on experience.

To create a function, start a new line with `def` (short for *definition*) followed by a space, and then a name of your own choosing followed by a pair of parentheses with no spaces before or inside. Then put a colon at the end of that line. For example, to create a simple function named `hello()`, type

```
def hello():
```

This is a function, but it doesn't do anything. To make the function do something, you have to write Python code on subsequent lines. To ensure that the new code is "inside" the function, indent each of those lines.


REMEMBER

Indentations matter big time in Python. There is no command that marks the end of a function. All indented lines below the `def` line are part of that function. The first un-indented line (indented as far out as the `def` line) is outside the function.

To make this function do something, put an indented line of code under `def`. We'll start by just having the function print `hello`. So, type **`print('Hello')`** indented under the `def` line. Now your code looks like this:

```
def hello():
    print('Hello')
```

If you run the code now, nothing will happen. That's okay. Nothing should happen because the code inside a function isn't executed until the functioned is *called*. You call your own functions the same way you call built-in functions: by writing code that calls the function by name, including the parentheses at the end.

For example, if you're following along, press Enter to add a blank line and then type **`hello()`** (no spaces in there) and make sure it's *not* indented. (You don't want this code to be indented because it's *calling* the function to execute its code; it's not *part of* the function.) So it looks like this:

```
def hello():
    print('Hello')

hello()
```

Still, nothing happens if you're in a Jupyter cell or a `.py` file because you've only typed the code so far. For anything to happen, you have to run the code in the

usual way in Jupyter or VS Code (if you're using a `.py` file in VS Code). When the code executes, you should see the output, which is just the word `Hello`, as shown in Figure 5-1.

FIGURE 5-1: Writing, and calling, a simple function named `hello()`.

Commenting a Function

Comments are always optional in code. But it's customary to make the first line under the `def` statement a *docstring* (text enclosed in triple quotation marks) that describes what the function does. It's also common to put a comment, preceded by a # sign, to the right of the parentheses in the first line. Here's an example using the simple `hello()` function:

```
def hello():   # Practice function
    """ A docstring describing the function """
    print('Hello')
```

Because they're just comments, they don't have any effect on what the code does. Comments are just notes to yourself or to programming team members describing what the code is about. Running the code again displays the same results.

As a bonus for VS Code users, when you start typing the function name, VS Code's IntelliSense help shows the `def` statement for your custom function as well as the docstring you typed for it, as shown in Figure 5-2. So you get to create custom help for your own custom functions.

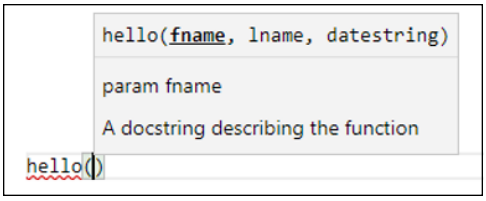


FIGURE 5-2: The docstring comment for your function appears in VS Code IntelliSense help.

Passing Information to a Function

You can pass information to a function for it to work on. To do so, enter a parameter name in the `def` statement for each piece of information you'll be passing to the function. You can use any name for the parameter, as long as it starts with a letter or underscore, followed by a letter, an underscore, or a number. The name should not contain spaces or punctuation. (Parameter names and variable names follow the same rules.) Ideally, the parameter should describe what's being passed in, for code readability, but you can use generic names like `x` and `y`, if you prefer.

Any name you provide as a parameter is local only to that function. For example, if you have a variable named `x` outside the function and another variable named `x` inside the function, any changes you make to the `x` variable inside the function won't affect the `x` variable outside the function.

The technical term for the way variables work inside functions is *local scope*, meaning the scope of the variables' existence and influence stays inside the function and does not extend further. Variables created and modified inside a function literally cease to exist the moment the function stops running, and any variables defined outside the function are unaffected by the goings-on inside the function. This is a good thing because when you're writing a function, you don't have to worry about accidentally changing a variable outside the function that happens to have the same name.

TECHNICAL STUFF

A function can *return* a value, and that returned value *is* visible outside the function. More on how this process works in a moment.

Suppose you want the `hello` function to say `hello` to whoever is using the app (and you have access to that information in some variable). To pass the information into the function and use it there, you would do the following:

- Put a parameter name inside the function's parentheses to act as a placeholder for the incoming information.
- Inside the function, use that name to work with the information passed in.

For example, suppose you want to pass a person's name into the `hello` function and then use the name in the `print()` statement. You could use any generic name for both the parameter and the function, like this:

```
def hello(x):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + x)
```

Inside the parentheses of `hello(x)`, the `x` is a parameter, a placeholder for whatever is being passed in. Inside the function, that `x` refers only to the value passed into the function. Any variables named `x` outside the function are separate from the `x` used in the parameter name and inside the function.

Generic names don't exactly help make your code easy to understand. It would be better to use a more descriptive name, such as `name` or even `user_name`, as in the following:

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)
```

In the `print()` function, we added a space after the *o* in *Hello* so there'd be a space between `Hello` and the name in the output.

When a function has a parameter, you have to pass it a value when you call it or it won't work. For example, if you added the parameter to the `def` statement and still tried to call the function without the parameter, as in the following code, running the code would produce an error:

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

hello()
```

The error would read something like the following:

```
hello() missing 1 required positional argument: 'user_name'
```

which is a major nerd-o-rama way of saying the `hello` function expected something to be passed into it.

For this particular function, a string needs to be passed. We know this because we concatenate (add) whatever is passed into the variable to another string (the word `hello` followed by a space). If you tried to concatenate a number to a string, you'd get an error.

The value you pass can be a literal (the exact data you want to pass in) or the name of a variable that contains that information. For example, when you run this code:

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

hello('Alan')
```

the output is `Hello Alan` because when you called the function with the following line of code, you passed `Alan` as a string:

```
hello('Alan')
```

You can use a variable to pass data too. For example, in the code in Figure 5-3 we stored the string `"Alan"` in a variable named `this_person`. Then we call the function using that variable name. Running that code produces `Hello Alan`, as shown at the bottom of that figure.

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

# Put a string in a variable named this_person.
this_person = 'Alan'
# Pass that variable name to the function.
hello(this_person)
```

```
Hello Alan
```

FIGURE 5-3: Passing data to a function via a variable.

Defining optional parameters with defaults

In the preceding section we mention that when you call a function that expects parameters without passing those parameters, you get an error. That was a little bit of a lie. You *can* write a function so that passing a parameter is optional, but you have to tell the function what to use if nothing gets passed. The syntax follows:

```
def functionname(parametername=defaultvalue):
```

The only thing that's really different is the `=defaultvalue` part after the parameter name. For example, you could rewrite the sample `hello()` function with a default value, like this:

```
def hello(user_name = 'nobody'):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)
```

Figure 5-4 shows the function after making that change, along with the output of testing the function.

First the code calls the function, passing it the value `Alan`:

```
hello('Alan')
```

So the output is

```
Hello Alan
```

The second line we used to test the function calls the function but doesn't pass in a value. In other words, it calls the function but with no value in the parentheses, like this:

```
hello()
```

Because this line doesn't pass in a value, the function defaults to `'nobody'` and the output, as you can see at the bottom of the figure, is

```
Hello nobody
```

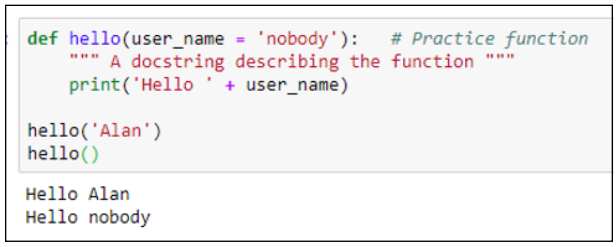


FIGURE 5-4: An optional parameter with a default value added to the `hello()` function.

Passing multiple values to a function

So far in all our examples we've passed just one value to the function. But you can pass as many values as you want. Just provide a parameter name for each value, and separate the names with commas.

For example, suppose you want to pass the user's first name, last name, and maybe a date to the function. You could define those three parameters like this:

```
def hello(fname, lname, datestring):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + fname + ' ' + lname)
    print('The date is ' + datestring)
```

Note that none of the parameters is optional. So when calling the function, you need to pass three values, such as this:

```
hello('Alan', 'Simpson', '12/31/2019')
```

Figure 5-5 shows an example of executing code with a hello() function that accepts three parameters.

```
def hello(fname, lname, datestring):   # Practice function
    """ A docstring describing the function """
    msg = "Hello " + fname + " " + lname
    msg += " you mentioned " + datestring
    print(msg)

hello('Alan', 'Simpson', '12/31/2019')
```

```
Hello Alan Simpson you mentioned 12/31/2019
```

FIGURE 5-5: The `hello` function with three parameters.

If you want to use some (but not all) optional parameters with multiple parameters, make sure the optional ones are the last ones entered. For example, consider the following, which would *not* work:

```
def hello(fname, lname='unknown', datestring):
```

If you try to run this code with that arrangement, you get an error that reads something along the lines of

```
SyntaxError: non-default argument follows default argument.
```

This error is trying to tell you that if you want to list both required parameters and optional parameters in a function, you have to put all the required ones first (in any order). Then the optional parameters can be listed after that with their = signs (in any order). So the following would work fine:

```
def hello(fname, lname, datestring=''):
    msg = 'Hello ' + fname + ' ' + lname
    if len(datestring) > 0:
        msg += ' you mentioned ' + datestring
    print(msg)
```

Logically, the code inside the function does the following

- Create a variable named `msg` and put in `Hello` and the first and last name.
- If the `datestring` passed has a length greater than `0`, add " `you mentioned` " and that `datestring` to the `msg` variable.
- Print whatever is in the `msg` variable at this point.

Figure 5-6 shows two examples of calling this version of the function. The first call passes three values, and the second call passes only two. Both work because the third parameter is optional. The output from the first call is the full output including the date, and the output from the second omits the part about the date.

```
def hello(fname, lname, datestring=''):
    msg = 'Hello ' + fname + ' ' + lname
    if len(datestring) > 0:
        msg += ' you mentioned ' + datestring
    print(msg)

hello('Alan', 'Simpson', '12/31/2019')
hello('Sammy', 'Schmeedledorp')
```

```
Hello Alan Simpson you mentioned 12/31/2019
Hello Sammy Schmeedledorp
```

FIGURE 5-6: Calling the `hello()` function with three parameters, and again with two parameters.

Using keyword arguments (kwargs)

If you've ever looked at the official Python documentation at Python.org, you may have noticed that they throw around the term *kwargs* a lot. That's short for *keyword arguments* and is yet another way to pass data to a function.

The term *argument* is the technical term for "the value you are passing to a function's parameters." So far, we've used strictly positional arguments. For example, consider these three parameters:

```
def hello(fname, lname, datestring=''):
```

When you call the function like this:

```
hello("Alan", "Simpson")
```

Python assumes `"Alan"` is the first name, because it's the first argument passed and `fname` is the first parameter in the function. `"Simpson"`, the second argument, is assumed to be `lname` because `lname` is the second parameter in the `def`

statement. The `datestring` is assumed to be empty because `datestring` is the third parameter in the `def` statement and nothing is being passed as a third argument.

As an alternative to relying solely on an argument's position in the code to associate it with a parameter name, you can tell the function what's what by using the syntax *`parameter = value`* in the code calling the function. For example, take a look at this call to `hello`:

```
hello(datestring='12/31/2019', lname='Simpson', fname='Alan')
```

When you run this code, it works fine even though the order of the arguments passed doesn't match the order of the parameter names in the `def` statement. But the order doesn't matter here because the parameter name that each argument goes with is included with the call. Clearly the `'Alan'` argument goes with the `fname` parameter because `fname` is the name of the parameter in the `def` statement.

The same concept applies if you pass variables. Again, the order doesn't matter. In the following example, the values to be passed to the function are first placed in variables named `attpt_date`, `last_name`, and so forth. Then the last line calls the `hello()` function again as in previous examples. But the value assigned to each parameter name is the name of a variable, not a literal value being passed in.

```
appt_date = '12/30/2019'
last_name = 'Janda'
first_name = 'Kylie'
hello(datestring=apt_date, lname=last_name, fname=first_name)
```

Figure 5-7 shows the result of running the code both ways. As you can see, it all works fine. There's no ambiguity about which argument goes with which parameter because the parameter name is specified in the calling code.

```
def hello(fname, lname, datestring):   # Practice function
    """ A docstring describing the function """
    msg = "Hello " + fname + " " + lname
    msg += " you mentioned " + datestring
    print(msg)

# Pass in in literal kwargs (identify each by parameter name)
hello(datestring='12/31/2019', lname='Simpson', fname='Alan')

# Pass in in kwargs from variables (identify each by parameter name)
appt_date = '12/30/2019'
last_name = 'Janda'
first_name = 'Kylie'
hello(datestring=appt_date, lname=last_name, fname=first_name)
```

```
Hello Alan Simpson you mentioned 12/31/2019
Hello Kylie Janda you mentioned 12/30/2019
```

FIGURE 5-7: Calling a function with keyword arguments (kwargs).

Passing multiple values in a list

So far we've been passing one piece of data at a time. But you can also pass iterables to a function. Remember an *iterable* is anything that Python can loop through to get values. A list is a simple and perhaps the most commonly used iterable.

The main trick to working with lists is this: If you want to alter the list contents (for example, by sorting the contents), make a copy of the list in the function and then make changes to the copy. You have to work with a copy of the list that was passed because the function doesn't receive the original list in a mutable (changeable) format; it receives only a pointer to the list, which indicates the list's location. Then the function can get the list's contents. The function can do anything it likes with its own copy of the list, but the original list remains unchanged.

After you have a copy of the list inside the function, you can sort that copy using the simple `sort()` method. Or, if you want to sort in descending order, use `sort(reverse=True)`.

For example, here is a new function named `alphabetize()` that takes one argument called `names`. The name of the parameter being passed in is `original_list`. The entire parameter declaration is `original_list=[]`. The square brackets indicate an empty list as the default, in case nothing is passed in as a parameter. In other words, we're using `=[]` to define the default input as an empty list. The function can alphabetize a list of any number of words or names:

```
def alphabetize(original_list=[]):
    """ Pass any list in square brackets, displays a string with items
    sorted """
    # Inside the function make a working copy of the list passed in.
    sorted_list = original_list.copy()
    # Sort the working copy.
    sorted_list.sort()
    # Make a new empty string for output
    final_list = ''
    # Loop through sorted list and append name and comma and space.
    for name in sorted_list:
        final_list += name + ', '
    # Knock off last comma space if the string is not blank
    final_list = final_list[:-2]
    # Print the alphabetized list.
    print(final_list)
```

The first line defines the function. Note that we used `original_list=[]`for the parameter. The default value (`=[]`) is optional, but we put it there so the function doesn't crash if you accidentally call it without passing in a list. Instead, it just creates an empty list. For example, when you start to type the function name in

VS Code, you get both the `def` statement and the docstring as IntelliSense help to remind you how to use the function, as in Figure 5-8.

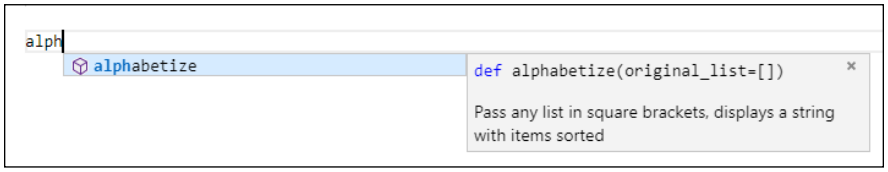


FIGURE 5-8: Using the `alphabetize` function in VS Code.

Because the function can't alter the list directly, it first makes a copy of the original list (the one that was passed) in a new list called `sorted_list`, with this line of code:

```
sorted_list = original_list.copy()
```

At this point, `sorted_list` isn't really sorted; it's still just a copy of the original. The next line of code does the sorting:

```
sorted_list.sort()
```

This function creates a string with the sorted items separated by commas. So the next line creates a new variable name, `final_list` and, after the `=` sign, starts the variable off as an empty string (two single quotation marks with no space between):

```
final_list = ''
```

This loop loops through the sorted list and adds each item in the list, separated by a comma and a space, to the `final_list` string:

```
for name in sorted_list:
    final_list += name + ', '
```

When that's done, if anything was added to `final_list`, it will have an extra comma and a space at the end. The following statement removes those last two characters, assuming the list is at least two characters in length:

```
final_list = final_list[:-2]
```

The next statement just prints `final_list` so you can see it.

To call the function, you can pass a list inside the parentheses of the function, like this:

```
alphabetize(['Schrepfer', 'Maier', 'Santiago', 'Adams'])
```

As always, you can also pass in the name of a variable that contains the list, as in this example:

```
names = ['Schrepfer', 'Maier', 'Santiago', 'Adams']
alphabetize(names)
```

Either way, the function displays those names in alphabetical order:

```
Adams, Maier, Santiago, Schrepfer
```

Passing in an arbitrary number of arguments

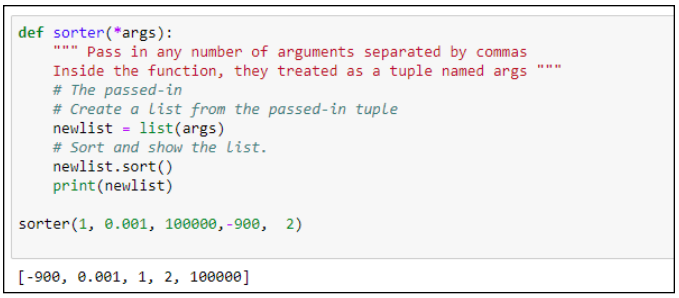
A list provides one way of passing a lot of values into a function. You can also design the function so that it accepts any number of arguments. Note that this method is not particularly faster or better, so use whichever is easiest or makes the most sense. To pass in any number of arguments, use `*args` as the parameter name, like this:

```
def sorter(*args):
```

Whatever you pass in becomes a tuple named `args` inside the function. Remember, a tuple is an immutable list (a list you can't change). So again, if you want to change things, you need to copy the tuple to a list and then work on that copy. Here is an example where the code uses the simple statement `newlist = list(args)`. You can read that as *the variable named newlist is a list of all the things that are in the args tuple*. The next line, `newlist.sort()` sorts the list, and `print` displays the contents of the list:

```
def sorter(*args):
    """ Pass in any number of arguments separated by commas
    Inside the function, they treated as a tuple named args. """

    # Create a list from the passed-in tuple.
    newlist = list(args)
    # Sort and show the list.
    newlist.sort()
    print(newlist)
```

Figure 5-9 shows an example of running this code with a series of numbers as arguments in a Jupyter cell. As you can see, the resulting list is in sorted order, as expected.



```
def sorter(*args):
    """ Pass in any number of arguments separated by commas
    Inside the function, they treated as a tuple named args """
    # The passed-in
    # Create a list from the passed-in tuple
    newlist = list(args)
    # Sort and show the list.
    newlist.sort()
    print(newlist)

sorter(1, 0.001, 100000,-900,  2)
```

```
[-900, 0.001, 1, 2, 100000]
```

FIGURE 5-9: A function accepting any number of arguments with `*args`.

Returning Values from Functions

So far, all our functions have displayed output on the screen so you can make sure the function works. In real life, it's more common for a function to *return* some value and put it in a variable specified in the calling code. The line that does the returning is typically the last line of the function followed by a space and the name of the variable (or some expression) that contains the value to be returned.

Here is a variation of the `alphabetize` function. It contains no `print` statement. Instead, at the end, it simply returns the alphabetized list (`final_list`) that the function created:

```
def alphabetize(original_list=[]):
    """ Pass any list in square brackets, displays a string with items
    sorted """
    # Inside the function make a working copy of the list passed in.
    sorted_list = original_list.copy()
    # Sort the working copy.
    sorted_list.sort()
    # Make a new empty string for output
    final_list = ''
    # Loop through sorted list and append name and comma and space.
    for name in sorted_list:
        final_list += name + ', '
    # Knock off last comma space
    final_list = final_list[:-2]
    # Return the alphabetized list.
    return final_list
```

The most common way to use functions is to store whatever they return in some variable. For example, in the following code, the first line defines a variable called `random_list`, which is just a list containing names in no particular order, enclosed in square brackets (which tells Python it's a list). The second line creates a new variable named `alpha_list` by passing `random_list` to the `alphabetize()` function and storing whatever that function returns. The final `print` statement displays whatever is in the `alpha_list` variable:

```
random_list = ['McMullen', 'Keaser', 'Maier', 'Wilson', 'Yudt', 'Gallagher',
   'Jacobs']
alpha_list = alphabetize(random_list)
print(alpha_list)
```

Figure 5-10 shows the result of running the whole kit-and-caboodle in a Jupyter cell.

```
def alphabetize(original_list=[]):
    """ Pass any list in square brackets, displays a string with items sorted """
    # Inside the function make a working copy of the list passed in.
    sorted_list = original_list.copy()
    # Sort the working copy.
    sorted_list.sort()
    # Make a new empty string for output
    final_list = ''
    # Loop through sorted list and append name and comma and space.
    for name in sorted_list:
        final_list += name + ', '
    # Knock of last comma space if final list is long enough
    final_list = final_list[:-2]
    # Print the alphabetized list.
    print(final_list)

names = ['McMullen', 'Keaser', 'Maier', 'Wilson', 'Yudt', 'Gallagher', 'Jacobs']
alphabetize(names)
```

```
Gallagher, Jacobs, Keaser, Maier, McMullen, Wilson, Yudt
```

FIGURE 5-10: Printing a string returned by the `alphabetize()` function.

Unmasking Anonymous Functions

Python supports the concept of *anonymous functions*, also called *lambda functions.* The *anonymous* part of the name is based on the fact that the function doesn't need to have a name (but *can* have one if you want it to). The *lambda* part is based on the use of the keyword *lambda* to define anonymous functions in Python. In other words, when you see the word `lambda` in Python code, that line of code is defining an anonymous function.

The minimal syntax for defining a lambda expression (with no name) follows:

```
lambda arguments : expression
```

Replace *arguments* with the data being passed into the expression. And replace *expression* with an expression (formula) that defines what you want the anonymous function to return.

A common example of using this syntax is when you're trying to sort strings of text when some of the names start with uppercase letters and some start with lowercase letters, as in these names:

```
Adams, Ma, diMeola, Zandusky
```

Suppose you write the following code to put the names in a list, sort the list, and then print it:

```
names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort()
print(names)
```

That output follows:

```
['Adams', 'Ma', 'Zandusky', 'diMeola']
```

Having diMeola come after Zandusky seems wrong to us and probably to you. But computers don't always see things the way we do. (Actually, they don't see anything because they don't have eyes or brains, but that's beside the point.) The reason diMeola comes after Zandusky is because the sort is based on ASCII, which is a system in which each character is represented by a number. All lowercase letters have numbers that are higher than uppercase numbers. So, when sorting, all the words starting with lowercase letters come after the words that start with an uppercase letter. If nothing else, this warrants at least a minor *hmm*.

To help with these matters, the Python `sort()` method lets you include a `key=` expression inside the parentheses, where you can tell it how to sort. The syntax is as follows:

```
.sort(key = transform)
```

The *transform* part is some variation on the data being sorted. If you're lucky and one of the built-in functions such as `len` (for *length*) will work, you can use that in place of *transform*, like this:

```
names.sort(key=len)
```

Unfortunately for us, the length of the string doesn't help with alphabetizing. So when you run this line of code, the order is

```
['Ma', 'Adams', 'diMeola', 'Zandusky']
```

The sort is going from the shortest string (the one with the fewest characters) to the longest string. Not helpful at the moment.

You can't write `key=lower` or `key=upper` to base the sort on all lowercase or all uppercase letters either, because `lower` and `upper` aren't built-in functions (which you can verify quickly by doing a web search for *python 3.7 built-in functions*).

In lieu of a built-in function, you can use a custom function that you define using `def`. For example, we can create a function named `lowercaseof()` that accepts a string and returns that string with all its letters converted to lowercase. Here is the function:

```
def lowercaseof(anystring):
    """ Converts string to all lowercase """
    return anystring.lower()
```

We made up the name `lowercaseof`, and `anystring` is a placeholder for whatever string you pass to it in the future. The line `return anystring.lower()` returns that string converted to all lowercase by using the `.lower()` method of the `str` (string) object.

TECHNICAL STUFF

Suppose you write this function in a Jupyter cell or `.py` file. Then you call the function with something like `print(lowercaseof('Zandusky'))`. What you get as output is the string converted to all lowercase, as in Figure 5-11.

```
: def lowercaseof(anystring):
      """ Converts string to all lowercase """
      return anystring.lower()

  print(lowercaseof('Zandusky'))

  zandusky
```

FIGURE 5-11: Putting a custom function named `lowercaseof()` to the test.

Okay, so now we have a custom function to convert any string to all lowercase letters. How do we use that as a sort key? Easy. Use `key=`*`transform`* the same as before, but replace *`transform`* with your custom function name. Our function

is named `lowercaseof`, so we'd use `.sort(key=lowercaseof)`, as shown in the following:

```
def lowercaseof(anystring):
    """ Converts string to all lowercase """
    return anystring.lower()

names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort(key=lowercaseof)
```

Running this code to display the list of names puts them in the correct order, because it based the sort on strings that are all lowercase. The displayed names are the same as before because only the sorting, which took place behind the scenes, used lowercase letters. The original data is still in its original uppercase and lowercase letters.

```
'Adams', 'diMeola', 'Ma', 'Zandusky'
```

If you're still awake and conscious after reading all this, you may be thinking, "Okay, you solved the sorting problem. But I thought we were talking about lambda functions here. Where's the lambda function?" There is no lambda function yet. But this is a perfect example of where you *could* use a lambda function, because the function you're calling, `lowercaseof()`, does all its work with just one line of code: `return anystring.lower()`.

When your function can do its thing with a simple one-line expression like that, you can skip the `def` and the function name and just use this syntax:

```
lambda parameters : expression
```

Replace *`parameters`* with one or more parameter names that you make up yourself (the names inside the parentheses after `def` and the function name in a regular function). Replace *`expression`* with what you want the function to return without the word `return`. So in this example, the key, using a lambda expression, would be

```
lambda anystring: anystring.lower()
```

Now you can see why it's an anonymous function. The entire first line with function name `lowercaseof()` has been removed. So the advantage of using the lambda expression is that you don't even need the external custom function. You just need the parameter followed by a colon and an expression that tells it what to return.

Figure 5-12 shows the complete code and the result of running it. You get the proper sort order without the need for a customer external function like `lower-caseof()`. You just use `anystring: anystring.lower()` (after `lambda`) as the sort key.

```
names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort(key = lambda anystring : anystring.lower())
print(names)
```

```
['Adams', 'diMeola', 'Ma', 'Zandusky']
```

FIGURE 5-12: Using a lambda expression as a sort key.

Note that `anystring` is a longer parameter name than most Pythonistas would use. Python folks are fond of short names, even single-letter names. For example, you could replace `anystring` with `s` (or any other letter), as in the following, and the code would work the same:

```
names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort(key=lambda s: s.lower())
print(names)
```

Way back at the beginning of this section we mentioned that `lambda` functions don't have to be anonymous. You can give them names and call them as you would other functions.

For example, here is a `lambda` function named `currency` that takes any number and returns a string in currency format (that is, with a leading dollar sign, commas between thousands, and two digits for pennies):

```
currency = lambda n: f"${n:,.2f}"
```

Here is one named `percent` that multiplies any number you send to it by 100 and displays it with two digits after the decimal point and a percent sign at the end:

```
percent = lambda n: f"{n:.2%}"
```

Figure 5-13 shows examples of both functions defined at the top of a Jupyter cell. Then a few `print` statements call the functions by name and pass some sample data to them. Each `print()` statement displays the number in the desired format.

```
# Show number in currency format.
currency = lambda n : f"${n:,.2f}"
# Show number in percent format.
percent = lambda n : f"{n:.2%}"

# Test currency function
print(currency(99))
print(currency(123456789.09876543))

# Test percent function
print(percent(0.065))
print(percent(.5))
```

```
$99.00
$123,456,789.10
6.50%
50.00%
```

FIGURE 5-13: Two anonymous functions for formatting numbers.

The reason you can define those two functions as single-line lambda expressions is because you can do all the work in one line, `f"${n:,.2f}"` for the first one and `f"{n:.2%}"` for the second one. But just because you *can* do it that way, doesn't mean you *must*. You could use regular functions too, as follows:

```
# Show number in currency format.
def currency(n):
    return f"${n:,.2f}"

def percent(n):
# Show number in percent format.
    return f"{n:.2%}"
```

With this longer syntax, you could pass in more information too. For example, you might default to a right-aligned format within a certain width (say 15 characters) so all numbers are right aligned to the same width. Figure 5-14 shows this variation of the two functions.

```
# Show number in currency format, specify width.
def currency(n, w=15):
    """ Show in currency format, width = 15 or width of your choosing """
    s = f"${n:,.2f}"
    # Pad left of output with spaces to width of w.
    return s.rjust(w)

# Show number in percent format, specify width.
def percent(n, w=15):
    """ Show in percent format, width = 15 or width of your choosing """
    # Show number in percent format.
    s = f"{n:.1%}"
    # Pad left of output with spaces to width of w.
    return s.rjust(w)
```

FIGURE 5-14: Two functions for formatting numbers with a fixed width.

In Figure 5-14, the second parameter is optional and defaults to `15` if omitted. So if you call the `currency()` function like this:

```
print(currency(9999))
```

you get $9,999.00 padding with enough spaces on the left to make the output 15 characters wide. If you call the `currency()` function like this instead:

```
print(currency(9999,20))
```

you get $9,999.00 padded with enough spaces on the left to make the output 20 characters wide.


TIP

The `.rjust()` method used in Figure 5-14 is a Python built-in string method that right justifies content by padding the left side of a string with sufficient spaces to make it the specified width. There's also an `.ljust()` method that left justifies output by padding the right side. Furthermore, you're not limited to adding blanks spaces. You can add any character you like instead of a space.

We find the whole business of `.ljust()` and `.rjust()` confusing at times. When in doubt, just do a web search for *python left justify* or *python right justify* to get the details.

So there you have it, the ability to create your own custom functions in Python. In real life, any time you find that you need access to the same chunk of code — the same bit of logic — over and over again in your app, don't simply copy and paste that chunk of code over and over. Instead, put the code in a function that you can call by name. That way, if you decide to change the code, you don't have to go digging through your app to find all the places that need changing. Just change it in the function where it's all defined in one place.

IN THIS CHAPTER

- Understanding classes and objects
- Learning how to create a class
- Initializing an object in a class
- Populating an object's attributes
- Discovering how to give a class methods
- Checking out class inheritance

Chapter 6
Doing Python with Class

In the preceding chapter, we talk about functions, which allow you to compartmentalize chunks of code that do specific tasks. In this chapter, you learn about classes, which allow you to compartmentalize code *and* data. You discover all the wonder, majesty, and beauty of classes and objects (okay, maybe we're overselling things a little there). But classes have become a defining characteristic of modern object-oriented programming languages such as Python.

We're aware we threw a whole lot of techno jargon your way in previous chapters. Don't worry. For the rest of this chapter we start off assuming that — like 99.9 percent of people in this world — you don't know a class from an object from a pastrami sandwich.

Mastering Classes and Objects

As you may know, Python is an object-oriented programming language. The concept of object-oriented programming (OOP) has been a major buzzword in the computer world for at least a couple decades. The term *object* stems from the fact that the model resembles objects in the real word in that each object is a thing that has certain attributes and characteristics that make it unique. For example, a chair is an object. Lots of different chairs exist that differ in size, shape, color, and material. But they're all still chairs.

How about cars? We all recognize a car when we see one. (Well, usually.) Even though cars aren't all exactly the same, they all have certain *attributes* (year, make, model, color) that make each unique. They have certain methods in common, where a *method* is an action or a thing the car can do. For example, cars all have go, stop, and turn actions that you control in pretty much the same way.

Figure 6-1 shows the concept where all cars (although not identical) have certain attributes and methods in common. In this case, you can think of the class Car as being a factory that creates all cars. After each car is created, it is an independent object. Changing one car has no effect on the other cars or the Car class.

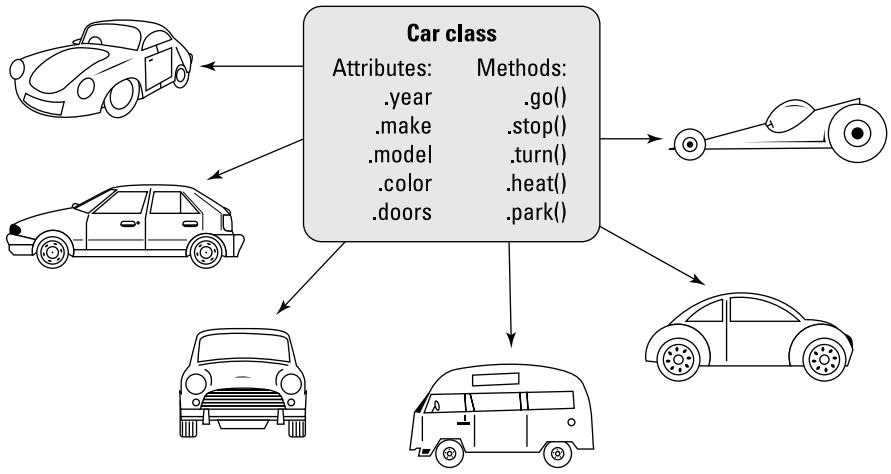


FIGURE 6-1: Different car objects.

If the factory idea doesn't work for you, think of a class as a type of blueprint. For instance, consider dogs. No, there's no physical blueprint for creating dogs, but there's dog DNA that does pretty much the same thing. The dog DNA can be considered a type of blueprint (like a Python class) from which all dogs are created. Dogs vary in attributes such as breed, color, and size, but they share certain behaviors (methods) such as eat and sleep. Figure 6-2 shows an example of a class of animal called Dog from which all dogs originate.

Even people can be viewed as objects in this manner. For example, perhaps you have a club and want to keep track of its members. Each member is a person, of course. But in code you can create a `Member` class to store information about each member. Each member would have certain attributes — username, full name, and so forth. You could also have methods such as `.archive()` to deactivate an account and `.restore()` to reactivate an account. The `.archive()` and `.restore()` methods are behaviors that let you control membership, in much the same way the accelerator, brake, and steering wheel allow you to control a car. Figure 6-3 shows the concept.

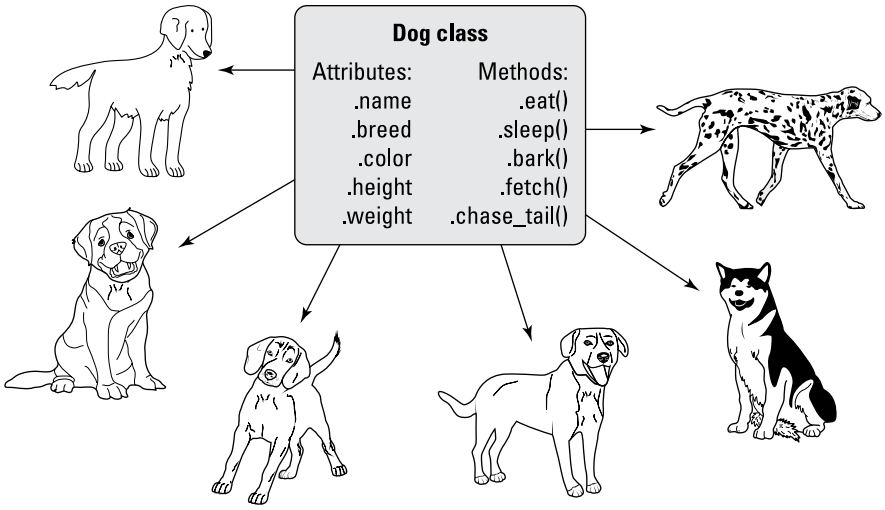


FIGURE 6-2: The Dog class creates many unique dogs.

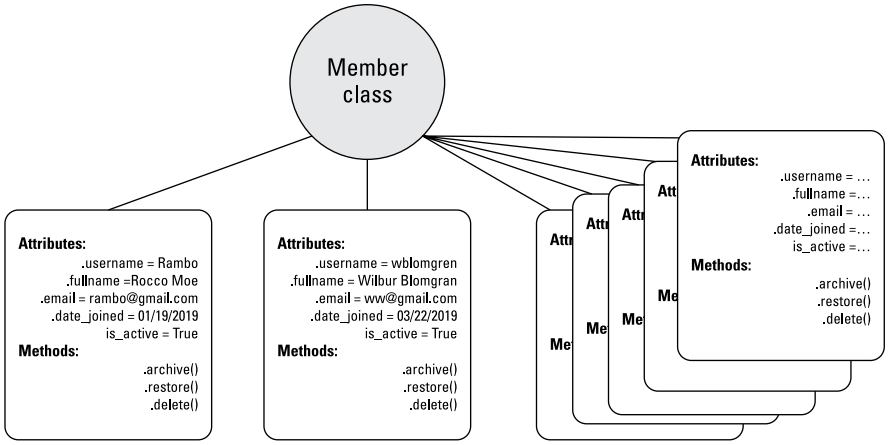


FIGURE 6-3: The `Member` class and member instances.

The main point is that each instance of a class is an independent object with which you can work. Changing one instance of a class has no effect on the class or on other instances, just as painting one car a different color has no effect on the car factory or on any other cars produced by that factory.

So, going back to initial concepts, all this business of classes and instances stems from a type of programming called *object-oriented programming* (*OOP* for short). Python, like any significant, serious, modern programming language, is

object-oriented. The main buzzwords you need to get comfortable with are the ones we've harped on in the last few paragraphs:

- **Class:** A piece of code from which you can generate a unique object, where each object is a single instance of the class. Think of a class as a blueprint or factory from which you can create individual objects.
- **Instance:** One unit of data plus code generated from a class as an instance of that class. Each instance of a class is also called an *object* just like all the different cars are objects, all created by some car factory (class).
- **Attribute:** A characteristic of an object that contains information about the object. Also called a *property* of the object. An attribute name is preceded by a dot, as in *`member.username`* which may contain the username for one site member.
- **Method:** A Python function associated with the class. A method defines an action that an object can perform. You call a method by preceding the method name with a dot and following it with a pair of parentheses. For example `member.archive()` might be a method that archives (deactivates) the member's account.

Creating a Class

You create your own classes like you create your own functions. You are free to name the class whatever you want, so long as it's a legitimate name that starts with a letter or underscore and contains no spaces or punctuation. It's customary to start a class name with an uppercase letter to help distinguish classes from variables. To get started, all you need is the word `class` followed by a space, a class name of your choosing, and a colon. For example, to create a new class named Member, use `class Member:`.

To make your code more descriptive, feel free to put a comment above the class definition. You can also put a docstring below the class line, which will show up whenever you type the class name in VS Code. For example, to add comments for your new `Member` class, you might type up the code like this:

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
```

That's it for defining a new class. However, it isn't useful until you specify what attributes you want each object that you create from this class to inherit from the class.

EMPTY CLASSES

If you start a class with `class` *`name`*`:` and then run your code before finishing the class, you'll actually get an error. To get around that, you can tell Python that you're just not quite ready to finish writing the class by putting the keyword `pass` below the definition, as in the following code:

```
# Define a new class name Member.
class Member:
    pass
```

In essence, what you're doing there is telling Python "Hey I know this class doesn't really work yet, but just let it pass and don't throw an error message telling me about it."

Creating an Instance from a Class

To grant to your class the capability to create instances (objects) for you, you give the class an `init` method. The word *init* is short for *initialize*. As a method, it's really just a function defined inside a class. But it must have the specific name `__init__` (that's two underscores followed by `init` followed by two more underscores).

TIP

That `__init__` is sometimes spoken as "*dunder init.*" The *dunder* part is short for *d*ouble *under*line.

The syntax for creating an `init` method is

```
def __init__(self[, suppliedprop1, suppliedprop2, ...])
```

The `def` is short for *define*, and `__init__` is the name of the built-in Python method that's capable of creating objects from within a class. The *`self`* part is just a variable name and is used to refer to the object being created at the moment. You can use the name of your own choosing instead of *`self`*. But *`self`* would be considered by most a best practice because it's explanatory and customary.

This business of classes is easier to learn and understand if you start simply. So, for a working example, you'll create a class named `Member`, into which you'll pass a username (`uname`) and full name (`fname`) whenever you want to create a member. As always, you can precede the code with a comment. You can also put

a docstring (in triple quotation marks) under the first line both as a comment but also as an IntelliSense reminder when typing code in VS Code:

```
# Define a class named Member for making member objects.
class Member:
    """ Create a member from uname and fname """
    def __init__(self, uname, fname):
```

When the `def __init__` line executes, you have an empty object named `self` inside the class. The `uname` and `fname` parameters hold whatever data you pass in; you see how that works in a moment.

An empty object with no data doesn't do you much good. What makes an object useful is its attributes: the information it contains that's unique to that object. So, in your class, the next step is to assign a value to each of the object's attributes.

Giving an Object Its Attributes

Now that you have a new, empty `Member` object, you can start giving it attributes and *populate* (store values in) those attributes. For example, let's say you want each member to have a `.username` attribute that contains the user's user name (perhaps for logging in). You have a second attribute named `fullname`, which is the member's full name. To define and populate those attributes, use the following:

```
self.username = uname
self.fullname = fname
```

The first line creates an attribute named `username` for the new instance (`self`) and puts into it whatever was passed into the `uname` attribute when the class was called. The second line creates an attribute named `fullname` for the new `self` object, and puts into it whatever was passed in as the `fname` variable. Add some comments and the entire class looks like this:

```
# Define a new class named Member.
class Member:
    """ Create a new member. """
    def __init__(self, uname, fname):
        # Define attributes and give them values.
        self.username = uname
        self.fullname = fname
```

Do you see what's happening? The `__init__` line creates a new empty object named `self`. Next, the `self.username = uname` line adds an attribute named `username` to the empty object, and puts into that attribute whatever was passed in as `uname`. Then the `self.fullname = fname` line does the same thing for the `fullname` attribute and the `fname` value that was passed in.

The convention for naming things in classes suggests using an initial cap for the class name. Attributes, however, should follow the standard for variables, which is all lowercase with an underscore to separate words within the name.

Creating an instance from a class

When you've created the class, you can create instances (objects) from it using this simple syntax:

```
this_instance_name = Member('uname', 'fname')
```

Replace *this_instance_name* with a name of your own choosing (in much the same way you may name a dog, who is an instance of the Dog class). Replace *uname* and *fname* with the username and full name you want to put into the object that will be created. Make sure you don't indent that code; otherwise, Python will think that new code still belongs to the class's code. It doesn't. It's new code to test the class.

So, for the sake of example, let's say you want to create a member named `new_guy` with the username `Rambo` and the full name `Rocco Moe`. Here's the code for that:

```
new_guy = Member('Rambo', 'Rocco Moe')
```

If you run this code and don't get any error messages, you know it at least ran. But to make sure, you can print the object or its attributes. To see what's really in the `new_guy` instance of `Members`, you can print it as a whole. You can also print just its attributes, `new_guy.username` and `new_guy.fullname`. You can also print `type(new_guy)` to ask Python what type `new_guy` is. This code does it all:

```
print(new_guy)
print(new_guy.username)
print(new_guy.fullname)
print(type(new_guy))
```

Figure 6-4 shows the code and the result of running it in a Jupyter cell.

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, uname, fname):
        # Define attributes and give them values.
        self.username = uname
        self.fullname = fname

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy
new_guy = Member('Rambo','Rocco Moe')

#See what's in the instance, as well as its individual properties.
print(new_guy)
print(new_guy.username)
print(new_guy.fullname)
print(type(new_guy))
```

```
<__main__.Member object at 0x000002175EA2E160>
Rambo
Rocco Moe
<class '__main__.Member'>
```

FIGURE 6-4: Creating a member from the `Member` class in a Jupyter cell.

In the figure, you can see that the first line of output is

```
<__main__.Member object at 0x000002175EA2E160>
```

This output tells you that `new_guy` is an object created from the `Member` class. The number at the end is its location in memory. Don't worry about that; you won't need to know about memory locations right now.

The next three lines of output are

```
Rambo
Rocco Moe
<class '__main__.Member'>
```

The first line is the username of `new_guy` (`new_guy.username`), and the second line is the full name of `new_guy` (`new_guy.fullname`). The last line is the type and tells you that `new_guy` is an instance of the `Member` class.

WARNING

Much as we hate to put any more burden on your brain cells right now, the words *object* and *property* are synonymous with *instance* and *attribute*. The `new_guy` instance of the `Member` class can also be called an object, and the `fullname` and `username` attributes of `new_guy` can also be called properties of that object.

Admittedly, it can be difficult to wrap your head around all these concepts, but just remember that an object is simply a handy way to encapsulate information about an item that's similar to other items (like all dogs are dogs and all cars are cars). What makes the item unique is its attributes, which won't necessarily be the same as the attributes of other objects of the same type, in much the same way that not all dogs are the same breed and not all cars are the same color.

We intentionally used `uname` and `fname` as parameter names to distinguish them from the attribute names `username` and `fullname`. However, this isn't a requirement. In fact, if anything, people tend to use the same names for the parameters as they do for the attributes.

Instead of `uname` for the parameter name, you can use `username` (even though it's the same as the attribute name). Likewise, you can use `fullname` in place of `fname`. Doing so won't alter how the class behaves. You just have to remember that the same name is being used in two different ways, first as a placeholder for data being passed into the class, and then later as an attribute name that gets its value from that passed-in value.

Figure 6-5 shows the same code as Figure 6-4 with `uname` replaced with `username` and `fname` replaced with `fullname`. Running the code produces the same output as before; using the same name for two different things didn't bother Python one bit.


```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy
new_guy = Member('Rambo','Rocco Moe')

#See what's in the instance, as well as its individual properties.
print(new_guy)
print(new_guy.username)
print(new_guy.fullname)
print(type(new_guy))
```

```
<__main__.Member object at 0x000002175EA2E240>
Rambo
Rocco Moe
<class '__main__.Member'>
```

FIGURE 6-5: The `Member` class with `username` and `fullname` for both parameters and attributes.

After you type a class name and the opening parenthesis in VS Code, its IntelliSense shows you the syntax for parameters and the first docstring in the code, as shown in Figure 6-6. Naming things in a way that's meaningful and including a descriptive docstring in the class makes it easier for you to remember how to use the class in the future.

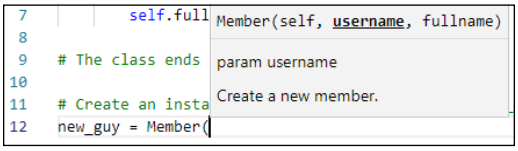


FIGURE 6-6: VS Code displays help when you access your own custom classes.

Changing the value of an attribute

When working with tuples, you can define *key*:*value* pairs, much like the *attribute*:*value* pairs you see here with instances of a class. There is one major difference, though: Tuples are immutable, meaning that after they're defined, your code can't change anything about them. This is not true with objects. After you create an object, you can change the value of any attribute at any time using the following simple syntax:

```
objectname.attributename = value
```

Replace *objectname* with the name of the object (which you've already created via the class). Replace *attributename* with the name of the attribute whose value you want to change. Replace *value* with the new value.

Figure 6-7 shows an example in which, after initially creating the `new_guy` object, the following line of code executes:

```
new_guy.username = "Princess"
```

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy
new_guy = Member('Rambo','Rocco Moe')

# See what's in the instance, as well as its individual properties.
print(new_guy.username)
print(new_guy.fullname)
print() #This just prints a blanks line.

# Change new_guy's user name then print both attributes again.
new_guy.username = "Princess"
print(new_guy.username)
print(new_guy.fullname)
```

```
Rambo
Rocco Moe

Princess
Rocco Moe
```

FIGURE 6-7: Changing the value of an object's attribute.

The lines of output under that show that `new_guy`'s `username` has indeed been changed to `Princess`. His full name hasn't changed because you didn't do anything to that in your code.

Defining attributes with default values

You don't have to pass in the value of every attribute for a new object. If you're always going to give an attribute some default value at the moment the object is created, you can just use `self.attributename = value`, the same as before, in which *attributename* is a name of your own choosing. And *value* can be some value you just set, such as `True` or `False` for a Boolean, or today's date, or anything that can be calculated or determined by Python without you providing the value.

For example, let's say that whenever you create a new member, you want to track the date you created that member in an attribute named `date_joined`. And you want to be able to activate and deactivate accounts to control user logins. So you create an attribute named `is_active` and decide to start a new member with that attribute set to `True`.

If you're going to be doing anything with dates and times, you'll want to import the `datetime` module, so put that at the top of your file, even before the `class Member:` line. Then you can add the following lines before or after the other lines that assign values to attributes within the class:

```
self.date_joined = dt.date.today()
self.is_active = True
```

Here is how you could add the `import` and those two new attributes to the class:

```
import datetime as dt

# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname

        # Default date_joined to today's date.
        self.date_joined = dt.date.today()
        # Set is active to True initially.
        self.is_active = True
```

WARNING

If you forget to import `datetime` at the top of the code, you'll get an error message when you run the code, telling you it doesn't know what `dt.date.today()` means. Just add the `import` line to the top of the code and try again.

There is no need to pass any new data into the class for the `date_joined` and `is_active` attributes because those attributes get default values from the code.

PERSISTING CHANGES TO DATA

What's the point of creating all these different classes and objects if everything just ceases to exist the moment the program ends? What does it mean to create a member if you can't store that information forever and use it to control members logging into a website or whatever?

Truthfully, creating objects that cease to exist the moment your program ends isn't the whole story. All the data you create and manage with classes and objects can be *persisted* (retained indefinitely) and be at your disposal at any time by storing that data in some kind of external file, usually a database.

We get to the business of persistent data in Book 3. But you need to learn the core Python basics first before you can understand more complicated topics such as persistence in data.

Note that a default value is just that: It's a value that is assigned automatically when you create the object. But you can change a default value in the same way you would change any other attribute's value, using this syntax:

```
objectname.attributename = value
```

For example, suppose you use the `is_active` attribute to determine whether a user is active and can log into your site. If a member turns out to be an obnoxious troll and you don't want him logging in anymore, you could just change the `is_active` attribute to `False` like this:

```
newmember.is_active = False
```

Giving a Class Methods

Any object you define can have any number of attributes, each given any name you like, to store information *about* the object, such as a dog's breed and color or a car's make and model. You can also define you own methods for any object, which are more like behaviors than facts about the object. For example, a dog can eat, sleep, and bark. A car can go, stop, and turn. A method is really just a function, as you learned in the preceding chapter. What makes it a method is the fact that it's associated with a particular class and with each specific object you create from that class.

Method names are distinguished from attribute names for an object by the pair of parentheses that follow the name. To define what the methods will be in your class, use this syntax for each method:

```
def methodname(self[, param1, param2, ...]):
```

Replace *methodname* with a name of your choosing (all lowercase, no spaces). Keep the word `self` in there as a reference to the object being defined by the class. Optionally, you can also pass in parameters after `self` using commas, as with any other function.

REMEMBER

Never type the square brackets (`[]`). They're shown here in the syntax only to indicate that parameter names after `self` are allowed but not required.

Let's create a method named `.show_date_joined()` that returns the user's name and the date the user joined in a formatted string. Here is how you could define this method:

```
# A method to return a formatted string showing date joined.
def show_datejoined(self):
    return f"{self.fullname} joined on {self.date_joined:%m/%d/%y}"
```

The name of the method is `show_datejoined`. The task of this method, when called, is to simply put together some nicely formatted text containing the member's full name and date joined.

To call the method from your code, use this syntax:

```
objectname.methodname()
```

Replace *objectname* with the name of the object to which you're referring. Replace *methodname* with the name of the method you want to call. Include the parentheses (no spaces). If the class's `__init__` method specifies only `self`, you don't pass anything in. However, if the `__init__` specifies additional parameters beyond `self`, you need to specify values for them. Figure 6-8 shows the complete example.

Note in Figure 6-8 how the `show_datejoined()` method is defined within the class. Its `def` is indented to the same level of the first `def`. The code that the method executes is indented under that. Outside the class, `new_guy = Member('Rambo', 'Rocco Moe')` creates a new member named `new_guy`. Then `new_guy.show_datejoined()` executes the `show_datejoined()` method, which in turn displays `Rocco Moe joined 11/18/20`, the day we ran the code.

```
import datetime as dt

# Define a new class named Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname
        # Default date_joined to today's date.
        self.date_joined =  dt.date.today()
        # Set is active to True initially.
        self.is_active = True

    # A method to return a formatted string showing date joined.
    def show_datejoined(self):
        return f"{self.fullname} joined on {self.date_joined:%m-%d-%y}"

    # Method to activate (True) or deactive (False) account.
    def activate(self, yesno):
        """ True for active, False to make inactive """
        self.is_active = yesno

# The class ends at the first unindented line.
# Create an instance of the Member class named new_guy.
new_guy = Member('Rambo','Rocco Moe')

# See what's in the instance, change is_active status.
print(new_guy.show_datejoined())
```

```
Rocco Moe joined on 11-18-20
```

FIGURE 6-8: Changing the value of an object's attributes.

Passing parameters to methods

You can pass data into methods in the same way you do functions: by using parameter names inside the parentheses. However, keep in mind that `self` is always the first name after the method name, and you never pass data to the `self` parameter. For example, let's say you want to create a method called `.activate()` and set it to `True` if the user is allowed to log in or `False` when the user isn't. Whatever you pass in is assigned to the `.is_active` attribute. Here's how to define that method in your code:

```
# Method to activate (True) or deactivate (False) account.
def activate(self, yesno):
    """ True for active, False to make inactive """
    self.is_active = yesno
```

The docstring is optional. However, the docstring would appear on the screen when you're typing relevant code in VS Code, so it would serve as a good reminder about what you can pass in. When executed, this method doesn't display anything on the screen; it just changes the `is_active` attribute for that member to whatever you passed in as the `yesno` parameter.

REMEMBER

It helps to understand that a method is really just a function. What makes a method different from a function is the fact that a method is always associated with some class. So a method is not as generic as a function.

Figure 6-9 shows the entire class followed by some code to test it. The line `new_guy = Member('Rambo', 'Rocco Moe')` creates a new member object named `new_guy`. Then `print(new_guy.is_active)` displays the value of the `is_active` attribute, which is `True` because that's the default for all new members.

```
import datetime as dt

# Define a new class named Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname
        # Default date_joined to today's date.
        self.date_joined =  dt.date.today()
        # Set is active to True initially.
        self.is_active = True

    # A method to return a formatted string showing date joined.
    def show_datejoined(self):
        return f"{self.fullname} joined on {self.date_joined:%m-%d-%y}"

    # Method to activate (True) or deactive (False) account.
    def activate(self, yesno):
        """ True for active, False to make inactive """
        self.is_active = yesno

# The class ends at the first unindented line.

# Create an instance of the Member class named new_guy.
new_guy = Member('Rambo','Rocco Moe')

# Is the new guy active?
print(new_guy.is_active)

# Try out the activate method.
new_guy.activate(False)

# Is the new guy still active?
print(new_guy.is_active)
```

```
True
False
```

FIGURE 6-9 Adding and testing an `.activate()` method.

The line `new_guy.activate(False)` calls the `activate()` method for that object and passes to it a Boolean `False`. Then `print(new_guy.is_active)` proves that the call to activate did indeed change the `is_active` attribute for `new_guy` from `True` to `False`.

Calling a class method by class name

As you've seen, you can call a class's method using the following syntax:

```
specificobject.method()
```

An alternative is to use the specific class name, which can help make the code easier for humans to understand:

```
Classname.method(specificobject)
```

Replace *Classname* with the name of the class (which we typically define starting with an uppercase letter), followed by the method name, and then put the specific object (which you've presumably already created) inside the parentheses.

For example, suppose we create a new member named `wilbur` using the `Member` class and this code:

```
wilbur = Member('wblomgren', 'Wilbur Blomgren')
```

Here, `wilbur` is the specific object we created from the `Member` class. We can call the `show_datejoined()` method on that object by using the syntax you've already seen:

```
print(wilbur.show_datejoined())
```

The alternative is to call the `show_datejoined()` method of the `Member` class and pass to it that specific object, `wilbur`, like this:

```
print(Member.show_datejoined(wilbur))
```

The output from both methods is the same (but with the date on which you ran the code):

```
Wilbur Blomgren joined on 11/18/20
```

The latter method isn't faster, slower, better, worse, or anything like that. It's just an alternative syntax you can use, and some people prefer it because starting the line with `Member` makes it clear to which class the `show_datejoined()` method belongs. This in turn can make the code more readable by other programmers or by yourself a year from now when you don't remember any of the things you wrote in the app.

Using class variables

So far you've seen examples of attributes, which are sometimes called *instance variables*, because they're placeholders that contain information that varies from one instance of the class to another. For example, in a Dog class, `dog.breed` may be `Poodle` for one dog but `Schnauzer` for another dog.

Another type of variable you can use with classes is called a *class variable*, which is applied to all new instances of the class that haven't been created yet. Class variables inside a class don't have any tie-in to `self` because the `self` keyword always refers to the specific object being created at the moment. To define a class variable, place the mouse pointer above the `def __init__` line and define the variable using the standard syntax:

```
variablename = value
```

Replace *`variablename`* with a name of your own choosing, and replace *`value`* with the specific value you want to assign to that variable. For example, let's say your code includes a `free_days` variable that grants people three months (90 days) of free access on sign-up. You're not sure if you want to commit to this forever, so rather than hardcode it into your app (so it's difficult to change), you can just make it a class variable that's automatically applied to all new objects, like this:

```
# Define a class named Member for making member objects.
class Member:
    """ Create a member object """
    free_days = 90

    def __init__(self, username, fullname):
```

Because we define the `free_days` variable before we define `__init__`, it's not tied to a specific object in the code.

Now suppose that later in the code, you want to store the date that the free trial expires. You could have an attribute named `date_joined` that represents the date that the member joined and another attribute named `free_expires` that represents the date that the user's free membership expires. You could determine the second date by adding the number of free days to the date the member joined. Intuitively, it may seem as though you could add `free_days` to the date using a simple syntax like this:

```
self.free_expires = dt.date.today() + dt.timedelta(days=free_days)
```

But if you tried to run this code, you'd get an error saying Python doesn't recognize the `free_days` variable name (even though it's defined right at the top of the class). Instead, you must precede the variable name with the class name or `self`. For example, this would work:

```
self.free_expires = dt.date.today() + dt.timedelta(days=Member.free_days)
```

Figure 6-10 shows the bigger picture. We removed some of the code from the original class to trim it and make it easier to focus on the new stuff. The `free_days = 365` line near the top sets the value of the `free_days` variable to 365. (We used 90 days in the previous example, but this is a new example, and we want to illustrate how the same code works with any number of days you specify in the `free_days` variable.) Then, later in the code, the `__init__` method uses `Member.free_freedays` to add that number of days to the current date. Running this code by creating a new member named `wilbur` and viewing his `date_joined` and `free_expires` attributes shows the current date (when you run the code) and the date 365 days after that.

```
import datetime as dt
# Define a new class name Member.
class Member:
    # Default number of free days.
    free_days = 365

    """ Create a new member. """
    def __init__(self, username, fullname):
        self.date_joined =  dt.date.today()
        # Set an expiration date
        self.free_expires = dt.date.today() + dt.timedelta(days = Member.free_days)

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy.
wilbur = Member('wblomgren', 'Wilbur Blomegren')

print(wilbur.date_joined)
print(wilbur.free_expires)
```

```
2020-11-18
2021-11-18
```

FIGURE 6-10: The `free_days` variable is a class variable in the `Member` class.

What if you later decide that giving people 90 free days is plenty. You could just change the 365 day value back to 90 in the class directly. Since it's a variable, you can do it on-the-fly, like this, outside the class:

```
#Set a default for free days.
Member.free_days = 90
```

When you run this code, you still create a user named `wilbur` with `date_joined` and `free_days` variables. But this time, `wilbur.free_expires` will be 90 days after the `datejoined`, not 365 days.

Using class methods

Recall that a method is a function that's tied to a particular class. So far, the methods you've used, such as `.show_datejoined()` and `.activate()`, have been

instance methods, because you always use them with a specific object — a specific instance of the class. With Python, you can also create class methods.

As the name implies, a *class method* is a method associated with the class as a whole, not specific instances of the class. In other words, class methods are similar in scope to class variables in that they apply to the whole class and not just individual instances of the class.

As with class variables, you don't need the `self` keyword with class methods because that keyword always refers to the specific object being created at the moment, not to all objects created by the class. So for starters, if you want a method to do something to the class as a whole, don't use `def` *`name`*`(self)` because the `self` immediately ties the method to one object.

It would be nice if all you had to do to create a class method is exclude the word `self`, but unfortunately it doesn't work that way. To define a class method, you first need to type this into your code:

```
@classmethod
```

The `@` at the start of this defines `classmethod` as a decorator — yep, yet another term to add to your ever-growing list of nerd-o-rama buzzwords. A *decorator* is generally something that alters or extends the functionality of that to which it is applied.

Below that line, define your class method using this syntax:

```
def methodname(cls,x, ...):
```

Replace *`methodname`* with the name you want to give your method. Leave the `cls` as-is because it's a reference to the class as a whole (because the `@classmethod` decorator defined it as such behind-the-scenes). After `cls`, you can have commas and the names of parameters that you want to pass to the method, just as you can with regular instance methods.

For example, suppose you want to define a method that sets the number of free days just before you start creating objects, so that all objects get the same `free_days` amount. The following code accomplishes that by first defining a class variable named `free_days` that has a given default value of `0`. (The default value can be anything.)

Further down in the class is this class method:

```
# Class methods follow @classmethod decorator and refer to cls rather than
# to self.
@classmethod
def setfreedays(cls,days):
    cls.free_days = days
```

This code tells Python that when someone calls the `setfreedays()` method on this class, it should set the value of `cls.free_days` (the `free_days` class variable for this class) to whatever number of days were passed in. Figure 6-11 shows a complete example in a Jupyter cell (which you can type and try for yourself), and the results of running that code.

```
import datetime as dt

# Define a new class named Member.
class Member:
    # Default number of free days.
    free_days = 365

    """ Create a new member. """
    def __init__(self, username, fullname):
        self.date_joined =  dt.date.today()
        # Set an expiration date.
        self.free_expires = dt.date.Today() + dt.timedelta(days - Member.free_days)

     # Class methods follow @classmethod decorator and refer to cls rather than to self.
    @classmethod
    def setfreedays(cls,days):
        cls.free_days = days
```

FIGURE 6-11: The `setfreedays()` method is a class method in the `Member` class.

REMEMBER

It's easy to forget that uppercase and lowercase letters matter a lot in Python, especially since it seems you're using lowercase 99.9 percent of the time. But as a rule, class names start with an initial cap, so any call to the class name must also start with an initial cap.

Using static methods

Just when you thought you may finally be finished learning about classes, it turns out there is another kind of method you can create in a Python class. It's called a static method and it starts with this decorator: `@staticmethod`.

So that part is easy. What makes a static method different from instance and class methods is that a *static method* doesn't relate specifically to an instance of an object or even to the class as a whole. It is a generic function, and the only reason

to define it as part of a class is if you want to use the same name elsewhere in another class in your code.

Wherever you want a static method, you type the `@staticmethod` line. Below that line, you define the static method like any other method, but you don't use `self` and you don't use `cls` because a static method isn't strictly tied to a class or an object. Here's an example of a static method:

```
@staticmethod
def currenttime():
    now = dt.datetime.now()
    return f"{now:%I:%M %p}"
```

So we have a method called `currenttime()` that isn't expecting any data to be passed in and doesn't care about the object or class you're working with. The method just gets the current datetime using `now = dt.datetime.now()` and then returns that information in a nice 12:00 PM format.

Figure 6-12 shows a complete example in which you can see the static method properly indented and typed near the end of the class. When code outside the class calls `Member.currenttime()`, it dutifully returns the time at the moment, without your having to say anything about a specific object from that class.

```
import datetime as dt

# Define a class named Member for making member objects.
class Member:
    # This is a class variable that's the same for all instances.
    free_days = 0

    """ Create a member object from username and fullname """
    def __init__(self, username, fullname):
        # Define properties and assign default values.
        self.datejoined = dt.date.today()
        self.free_expires = dt.date.today() + dt.timedelta(Member.free_days)

    # Class methods follow @classmethods and use cls rather than self.
    @classmethod
    def setfreedays(cls,days):
        cls.free_days = days

    @staticmethod
    def currenttime():
        now = dt.datetime.now()
        return f"{now:%I:%M %p}"

# Class definition ends at last indented line

# Try out the new static method (no object required)
print(Member.currenttime())
```

```
03:24 PM
```

FIGURE 6-12: The `Member` class now has a static method named `currenttime()`.

Understanding Class Inheritance

People who are into object-oriented programming live to talk about class inheritance and subclasses and so on, stuff that means little or nothing to the average Joe or Josephine on the street. Still, what they're talking about as a Python concept is something you see in real life all the time.

As mentioned, if we consider dog DNA to be a kind of factory or Python class, we can lump all dogs together as members of a class of animals we call dogs. Even though each dog is unique, all dogs are still dogs because they are members of the class we call dogs, and we can illustrate that, as in Figure 6-13.

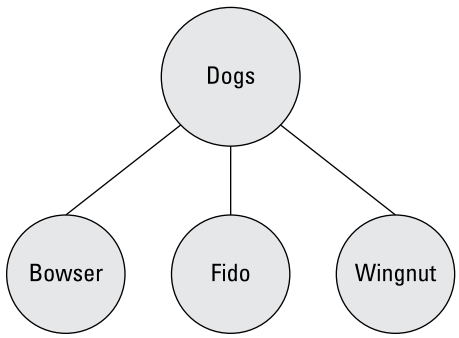


FIGURE 6-13: Dogs as objects of the class dogs.

So each dog is unique (although no other dog is as good as yours), but what makes dogs similar to one another are the characteristics they *inherit* from the class of dogs.

The notions of class and class inheritance that Python and other object-oriented languages offer didn't materialize out of the clear blue sky just to make it harder and more annoying to learn this stuff. Much of the world's information can best be stored, categorized, and understood by using classes and subclasses and sub-subclasses, on down to individuals.

For example, you may have noticed that other dog-like creatures roam the planet (although they're probably not the kind you'd like to keep around the house as pets). Wolves, coyotes, and jackals come to mind. They are similar to dogs in that they all *inherit* their dogginess from a higher-level class we could call canines, as shown in Figure 6-14.

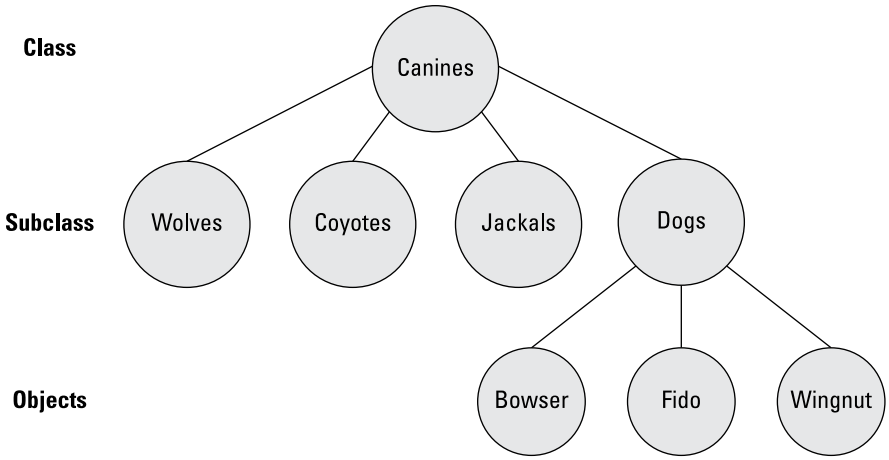


FIGURE 6-14: Several different kinds of animals are similar to dogs.

Using our dog analogy, we certainly don't need to stop at canines on the way up. We can put mammals above that, because all canines are mammals. We can put animals above that, because all mammals are animals. And we can put living things above that, because all animals are living things. So basically all the things that make a dog a dog stem from the fact that each *inherits* certain characteristics from numerous classes, or critters, that preceded it.

TECHNICAL STUFF

To the biology brainiacs out there, yes we know that Mammalia is a class, Canis is a genus, and below that are species. So you don't need to email or message us on that. We're using *class* and *subclass* terms here just to relate the *concept* to classes, subclasses, and objects in Python.

Obviously the concept doesn't apply just to dogs. The world has lots of different cats too. There's cute little Bootsy, with whom you'd be happy to share your bed, and plenty of other felines, such as lions, tigers, and jaguars, with whom you probably wouldn't.

TIP

If you do a web search for *living things hierarchy* and click Images, you'll see just how many ways there are to classify all living things, and how inheritance works its way down from the general to the specific living thing.

Even our car analogy can follow along with this. At the top, we have transportation vehicles. Under that, perhaps boats, planes, and automobiles. Under automobiles we have cars, trucks, vans, and so forth and so on, down to any one specific car. So classes and subclasses are nothing new. What's new is simply thinking about representing those things to mindless machines that we call computers. So let's see how you would do that.

From a coding perspective, the easiest way to do inheritance is to create subclasses within a class. The class defines things that apply to all instances of that class. Each subclass defines things relevant only to the subclass without replacing anything that's coming from the generic parent class.

Creating the base (main) class

Subclasses inherit all the attributes and methods of some higher-level main class, or parent class, which is usually referred to as the *base class*. This class is just any class, no different from what you've seen in this chapter so far. We'll use a `Member` class again, but we'll whittle it down to some bare essentials that have nothing to do with subclasses, so you don't have to dig through irrelevant code. Here is the basic class:

```
# Class is used for all kinds of people.
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class attributes and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)
```

By default, new accounts expire in one year. So this class first sets a class variable name `expiry_days` to `365` to be used in later code to calculate the expiration date from today's date. As you'll see later, we used a class variable to define `expiry_days` because we can give it a new value from a subclass.

To keep the code example simple and uncluttered, this version of the `Member` class accepts only two parameters, `firstname` and `lastname`.

Figure 6-15 shows an example of testing the code with a hypothetical member named Joe. Printing Joe's `firstname`, `lastname`, and `expiry_date` shows what you would expect the class to do when passing the `firstname Joe` and the `lastname Anybody`. When you run the code, the `expiry_date` should be one year from the date when you run the code.

```
import datetime as dt
# Class is used for all kinds of people.
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class attributes and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)

# Outside the class now.
Joe = Member('Joe', 'Anybody')
print(Joe.firstname)
print(Joe.lastname)
print(Joe.expiry_date)
```

```
Joe
Anybody
2019-12-08
```

FIGURE 6-15: A simplified `Member` class.

Now suppose our real intent is to make two different kinds of users, `Admins` and `Users`. Both types of users will have the attributes that the `Member` class offers. So by defining those types of users as subclasses of `Member`, they will automatically get the same attributes (and methods, if any).

Defining a subclass

To define a subclass, make sure you get the cursor below the base class and back to no indentation, because the subclass isn't a part of, or contained within, the base class. To define a subclass, use this syntax:

```
class subclassname(mainclassname):
```

Replace *`subclassname`* with whatever you want to name this subclass. Replace *`mainclassname`* with the name of the base class, as defined at the top of the base class. For example, to make a subclass of `Member` named `Admin`, use the following:

```
class Admin(Member):
```

To create another subclass named `User`, add this code:

```
class User(Member):
```

If you leave the classes empty, you won't be able to test them because you'll get an error message telling you the class is empty. But you can put the word `pass` as the first command in each one. This is your way of telling Python "Yes I know these classes are empty, but let it pass, don't throw an error message." You can

put a comment above each one to remind you of what each one is for, as in the following:

```
# Subclass for Admins.
class Admin(Member):
    pass

# Subclass for Users.
class User(Member):
    pass
```

When you use the subclasses, you don't have to make any direct reference to the `Member` class. `Admin`s and `User`s will both inherit all the `Member` stuff automatically. So, for example, to create an `Admin` named `Annie`, you'd use this syntax:

```
Ann = Admin('Annie', 'Angst')
```

To create a `User`, do the same thing with the `User` class and a name for the user. For example:

```
Uli = User('Uli', 'Ungula')
```

To see if this code works, you can do the same thing you did for member `Ann`. After you create the two accounts, use `print()` statements to see what's in them. Figure 6-16 shows the results of creating the two users. `Ann` is an `Admin`, and `Uli` is a `User`, but both automatically get all the attributes assigned to members. (The `Member` class is directly above the code shown in the image. We left that out because it hasn't changed.)

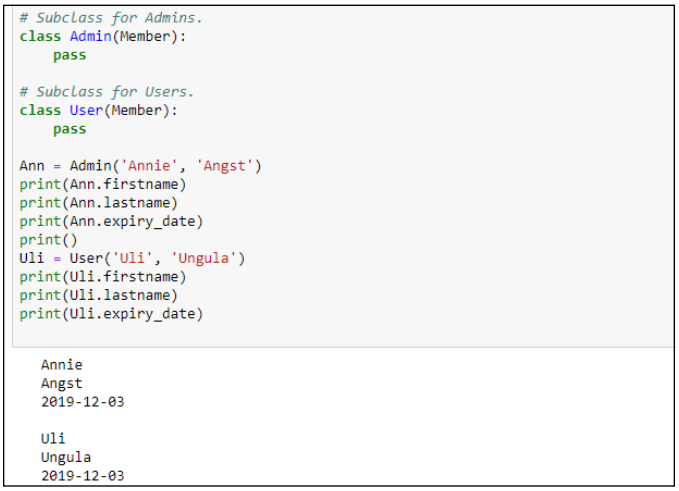


FIGURE 6-16: Creating and testing the `Admin` and `User` classes.

So what you've learned here is that the subclass accepts all the different parameters that the base class accepts and assigns them to attributes, same as the `Person` class. But so far, `Admin` and `User` are just members with no unique characteristics. In real life, there will probably be some differences between these two types of users. In the next sections, you learn ways to make these differences happen.

Overriding a default value from a subclass

One of the simplest things you can do with a subclass is to give an attribute that has a default value in the base class some other value. For example, in the `Member` class we created a variable named `expiry_days` to be used later in the class to calculate an expiration date. But suppose you want `Admin` accounts to never expire (or to expire after some ridiculous duration so there's still some date there). Simply set the new `expiry_date` in the `Admin` class (and you can remove the `pass` line because the class won't be empty anymore). Here's how these changes might look in your `Admin` subclass:

```
# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100
```

Whatever value you pass will override the default set near the top of the `Member` class and will be used to calculate the `Admin`'s expiration date.

Adding extra parameters from a subclass

Sometimes members of a subclass have a parameter value that other members don't. In that case, you may want to pass a parameter from the subclass that doesn't exist in the base class. Doing so is a little more complicated than just changing a default value, but it's a common technique so you should be aware of it. Let's work through an example.

For starters, your subclass will need its own `def __init__` line that contains everything that's in the base class's `__init__`, plus any extra stuff you want to pass. For example, let's say admins have some secret code and you want to pass that from the `Admin` subclass. You still have to pass the first and last name, so your `def __init__` line in the `Admin` subclass will look like this:

```
def __init__(self, firstname, lastname, secret_code):
```

The indentation level will be the same as the lines above it.

Next, any parameters that belong to the base class, `Member`, need to be passed up there using this rather odd-looking syntax:

```
super().__init__(param1, param2, ...)
```

Replace *param1*, *param2*, and so forth with the names of parameters you want to send to the base class. The information you're providing in the parameters should be everything that's already in the `Member` parameters excluding `self`. In this example, `Member` expects only `firstname` and `lastname`, so the code for this example is

```
super().__init__(firstname, lastname)
```

Whatever you didn't provide in the first set of parameters, you can assign to the subclass object using this code:

```
self.secret_code = parametername
```

Replace *parametername* with the name of the parameter that you didn't send up to `Member`. In this case, that would be the `secret_code` parameter. So the code would be:

```
self.secret_code = secret_code
```

Figure 6-17 shows an example in which we created an `Admin` user named `Ann` and passed `PRESTO` as her secret code. Printing all her attributes shows that she does indeed still have the right expiration date plus a secret code. As you can see, we also created a regular `User` named `Uli`. `Uli`'s data isn't affected by the changes to `Admin`.

In our working example, we haven't given regular users a secret code yet. If you try to print a regular user with the Python code as shown, you'll get an error because that Python code isn't yet written to accommodate users that have no secret code.

One solution is to just remember that regular users don't have a secret code. So when using the app, never try to print the secret code for a regular user. But it would be better if the code handled the error gracefully for us. To do so, we would ensure that every user's account is associated with a secret code. For regular users, the secret code will be empty, which prevents them from accessing administrator information. Only admins would have valid secret codes.

```
# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100
    # Subclass parameters
    def __init__(self, firstname, lastname, secret_code):
        # Pass Member parameters on up to Member class.
        super().__init__(firstname, lastname)
        # Assign the remaining parameter to this object.
        self.secret_code = secret_code

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.firstname, Ann.lastname, Ann.expiry_date, Ann.secret_code)

print()
Uli = User('Uli', 'Ungula')
print(Uli.firstname, Uli.lastname, Uli.expiry_date)
```

```
Annie Angst 2118-12-03 PRESTO

Uli Ungula 2019-12-03
```

FIGURE 6-17: The `Admin` subclass has a new `secret_code` parameter.

Should a member join as a regular member and later become an admin, the Python code need only change the empty `secret_code` to a valid `secret_code`.

If your class assigns a `secret_code` to all users (not just admins), you won't get an error when you print the data for a regular user. Instead, the secret code for a regular user will appear as a blank space. To assign a secret code to every member, even when that secret code is blank, add the following to the main `Member` class:

```
# Default secret code is nothing
self.secret_code = ""
```

So even though you don't do anything with `secret_code` in the `User` subclass, you don't have to worry about throwing an error if you try to access the secret code for a `User`. The `User` will have a secret code, but it will just be an empty string. Figure 6-18 shows all the code with both subclasses, and also an attempt to print `Uli.secret_code`, which just displays nothing without throwing an error message.

We left the `User` subclass with `pass` as its only statement. In real life, you would probably come up with more default values or parameters for your other subclasses. But the syntax and code is the same for all subclasses. The skills you've learned in this section will work for all your classes and subclasses.

```
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class properties and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Properties (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)
        # Default secret code is nothing
        self.secret_code = ''

    # Method in the base class
    def showexpiry(self):
        return  f"{self.firstname} {self.lastname} expires on {self.expiry_date}"

# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100

    # Subclass parameters
    def __init__(self, firstname, lastname, secret_code):
        # Pass Member parameters on up to Member class.
        super().__init__(firstname, lastname)
        # Assign the remaining parameter to this object.
        self.secret_code = secret_code

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.firstname, Ann.lastname, Ann.expiry_date, Ann.secret_code)
print() # Add a blank line to output.

Uli = User('Uli', 'Ungula')
print(Uli.firstname, Uli.lastname, Uli.expiry_date, Uli.secret_code)
```

```
Annie Angst 2118-12-08 PRESTO

Uli Ungula 2019-12-08
```

FIGURE 6-18: The complete `Admin` and `User` subclasses.

Calling a base class method

Methods in the base class work the same for subclasses as they do for the base class. To try out a method in the base class, add a new method called `showexpiry(self)` to the bottom of the base class, as follows:

```
class Member:
    """ The Member class attributes and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)
```

```
        # Default secret code is nothing
        self.secret_code = ''

    # Method in the base class.
    def showexpiry(self):
       return  f"{self.firstname} {self.lastname} expires on {self.expiry_date}"
```

The `showexpiry()` method, when called, returns a formatted string containing the user's first and last name and expiration date. Leaving the subclasses untouched and executing the code displays the names and expiry dates of `Ann` and `Uli`:

```
Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.showexpiry())


Uli = User('Uli', 'Ungula')
print(Uli.showexpiry())
```

Here is that output, although your dates will differ based on the date you ran the code:

```
Annie Angst expires on 2118-12-04
Uli Ungula expires on 2019-12-04
```

Using the same name twice

You may be wondering about what happens when you use the same name more than once? Python will always opt for the most specific one, the one tied to the subclass. It will use the more generic method from the base class only if nothing in the subclass has that method name.

To illustrate, here's some code that defines a `Member` class with just a few attributes and methods, to get any irrelevant code out of the way. Comments in the code describe what's going on in the code:

```
class Member:
    """ The Member class attributes and methods """
    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname

    # Method in the base class
    def get_status(self):
```

```
        return  f"{self.firstname} is a Member."

# Subclass for Administrators
class Admin(Member):
    def get_status(self):
        return  f"{self.firstname} is an Admin."

# Subclass for regular Users
class User(Member):
    def get_status(self):
        return  f"{self.firstname} is a regular User."
```

The `Member` class, and both the `Admin` and `User` classes, have a method named `get_status()`, which shows the member's first name and status. Figure 6-19 shows the result of running that code with an `Admin`, a `User`, and a `Member` who is neither an `Admin` nor a `User`. As you can see, the `get_status` called in each case is the `get_status()`associated with the person's subclass (or base class in the case of the person who is a `Member` but neither an `Admin` or `User`).

```
: class Member:
      """ The Member class attributes and methods are for everyone """
      # Initialize a member object.
      def __init__(self, firstname, lastname):
          # Attributes (instance variables) for everybody.
          self.firstname = firstname
          self.lastname = lastname

      # Method in the main class
      def get_status(self):
          return  f"{self.firstname} is a Member."

  # Subclass for Administrators
  class Admin(Member):
      def get_status(self):
          return  f"{self.firstname} is an Admin."

  # Subclass for regular Users
  class User(Member):
      def get_status(self):
          return  f"{self.firstname} is a regular User."

  # Create an admin
  Ann = Admin('Annie', 'Angst')
  print(Ann.get_status())

  #Create a user
  Uli = User('Uli', 'Ungula')
  print(Uli.get_status())

  # Create a member (neither Admin or User)
  Manny = Member("Mindy", "Membo")
  print(Manny.get_status())

  Annie is an Admin.
  Uli is a regular User.
  Mindy is a Member.
```

FIGURE 6-19: Three methods with the same name, `get_status()`.

Python has a built-in `help()` method that you can use with any class to get more information about that class. For example, at the bottom of the code in Figure 6-19, add this line:

```
help(Admin)
```

When you run the code again, you'll see some information about that Admin class, as shown in Figure 6-20.

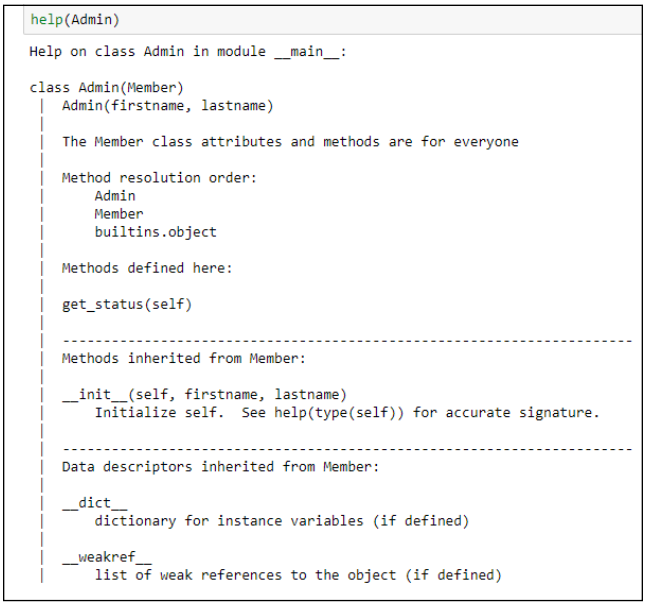


FIGURE 6-20: Output from `help(Admin)`.

You don't need to worry about all the details in Figure 6-20 right now. The most important section is the one titled *Method resolution order*, which looks like this:

```
Method resolution order:
     Admin
     Member
     builtins.object
```

The method resolution order tells you that if a class (and its subclasses) all have methods with the same name (such as `get_status`), a call to `get_status()` from an `Admin` user will cause Python to look in `Admin` for that method and, if it exists, use it. If no `get_status()` method was defined in the `Admin` subclass, Python looks in the `Member` class and uses that one, if found. If neither of those had a `get_status` method, it looks in `builtins.object`, which is a reference to certain built-in methods that all classes and subclasses share.

So the bottom line is, if you do store your data in hierarchies of classes and subclasses, and you call a method on a subclass, Python will use that subclass method if it exists. If not, Python will use the base class method if it exists. If that also doesn't exist, it will try the built-in methods. And if all else fails, it will throw an error because it can't find the method your code is trying to call. Usually the main reason for this type of error is that you simply misspelled the method name in your code, so Python can't find it.

An example of a built-in method is `__dict__`. The `dict` is short for *dictionary,* and those are double-underscores surrounding the abbreviation. Referring to Figure 6-20, executing the following command:

```
print(Admin.__dict__)
```

doesn't cause an error, even though we've never defined a method named `__dict__`. That's because there is a built-in method with that name, and when called with `print()`, it shows a dictionary of methods (both yours and built-in ones) for that object. The method resolution order isn't something you have to get too involved with this early in the learning curve. Just be aware that if you try to call a method that doesn't exist at any of those three levels, such as this:

```
print(Admin.snookums())
```

you will get an error that looks something like this:

```
---> print(Admin.snookums())
AttributeError: type object 'Admin' has no attribute 'snookums'
```

This error is telling you that Python has no idea what `snookums()` is about. As mentioned, in real life, this kind of error is usually caused by misspelling the method name in your code.

Classes (and to some extent, subclasses) are heavily used in the Python world, and what you've learned here should make it easy to write your own classes, as well as to understand classes written by others. You'll want to learn one more core Python concept before you finish this book: how Python handles errors, and things you can do in your own code to better handle errors.

IN THIS CHAPTER

- Discovering exceptions
- Finding out how to handle errors gracefully
- Making sure your app doesn't crash
- Checking out `try...except...else...finally`
- Learning to raise your own exceptions

Chapter 7
Sidestepping Errors

We all want our programs to run perfectly all the time. But sometimes, situations in the real world stop a program from running. The problem isn't with you or your program. Usually, the person using the program did something wrong. Error handling is all about anticipating these problems, catching the error, and then informing users of the problem so they can fix it.

The techniques we describe here aren't for fixing bugs in your code. You have to fix that type of error yourself. We're talking strictly about errors in the environment in which the program is running, over which you have no control. *Handling* the error is simply a way of replacing the tech-speak error message that Python normally displays, which is meaningless to most people, with a message that tells them in plain English what's wrong and, ideally, how to fix it.

Again, the user will be fixing *the environment in which the program* is running — they won't be fixing your code.

Understanding Exceptions

In Python (and all other programing languages) the term *exception* refers to an error that isn't due to a programming error. Rather, it's an error in the real world that prevents the program from running properly. As a simple example, let's have your Python app open a file. The syntax for that is easy:

```
name = open(filename)
```

Replace *name* with a name of your own choosing, same as a variable name. Replace *filename* with the name of the file. If the file is in the same folder as the code, you don't need to specify a path to the folder because the current folder is assumed.

Figure 7-1 shows an example. We used VS Code for this example so that you can see the contents of the folder in which we worked. The folder contains a file named `showfilecontents.py`, which is the file that contains the Python code we wrote. The other file is named `people.csv`.

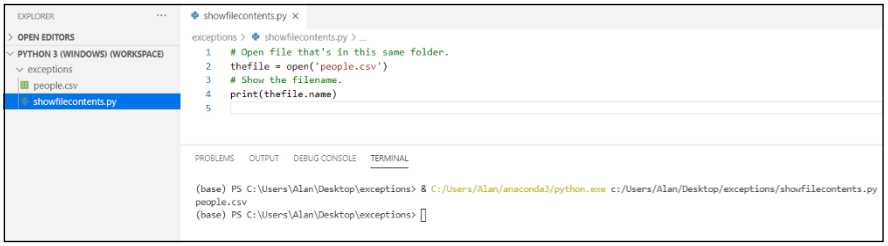


FIGURE 7-1: The `showfilecontents.py` and `people.csv` files in a folder in VS Code.

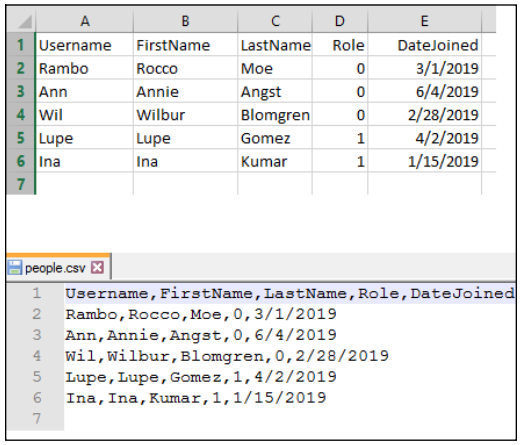
The `showfilecontents` file contains code. The `people.csv` file contains data (information about people). Figure 7-2 shows the content of the `people.csv` file in Excel (top) so it's easy for you to read and in a text editor (bottom), which is how it looks to Python and other languages. The file's content doesn't matter much right now; what you're learning here will work in any external file.

The Python code is just two lines (excluding the comments), as follows:

```
# Open file that's in this same folder.
thefile = open('people.csv')
# Show the filename.
print(thefile.name)
```

The first line of code opens the file named `people.csv`. The second line of code displays the filename (`people.csv`) on the screen. Running that simple `showfilecontents.py` file (by right-clicking its name in VS Code and choosing Run

Python File in Terminal) displays `people.csv` on the screen — assuming a file named `people.csv` exists in the folder to open. This assumption is where exception handling comes in.



	A	B	C	D	E
1	Username	FirstName	LastName	Role	DateJoined
2	Rambo	Rocco	Moe	0	3/1/2019
3	Ann	Annie	Angst	0	6/4/2019
4	Wil	Wilbur	Blomgren	0	2/28/2019
5	Lupe	Lupe	Gomez	1	4/2/2019
6	Ina	Ina	Kumar	1	1/15/2019
7					

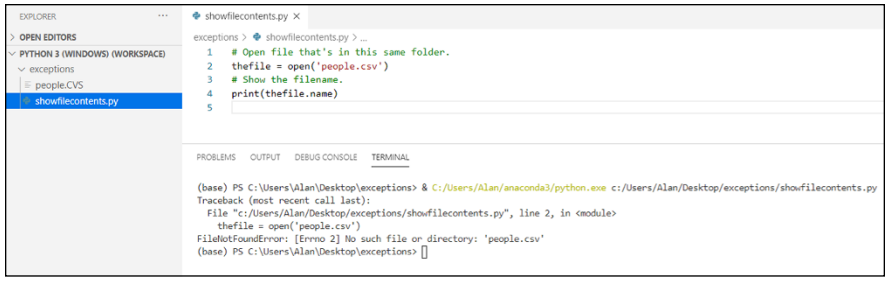
people.csv

```
1  Username,FirstName,LastName,Role,DateJoined
2  Rambo,Rocco,Moe,0,3/1/2019
3  Ann,Annie,Angst,0,6/4/2019
4  Wil,Wilbur,Blomgren,0,2/28/2019
5  Lupe,Lupe,Gomez,1,4/2/2019
6  Ina,Ina,Kumar,1,1/15/2019
7
```

FIGURE 7-2: The contents of the `people.csv` file in Excel (top) and a text editor (bottom).

Suppose that for reasons beyond your control, the `people.csv` file isn't there because some person or automated procedure failed to put it there. Or perhaps someone misspelled the filename. It's easy to accidentally type, say, `.cvs` rather than `.csv` for the filename. Running the app *raises an exception* (which in English means "displays an error message"), as you can see in the Terminal window at the bottom of Figure 7-3. The exception reads

```
Traceback (most recent call last):
  File "c:/ Users/ acsimpson/ Desktop/ exceptions/ showfilecontents.py", line 2,
 in <module>
    thefile = open('people.csv')
FileNotFoundError: [Errno 2] No such file or directory: 'people.csv'
```



EXPLORER

OPEN EDITORS

PYTHON 3 (WINDOWS) (WORKSPACE)

exceptions

people.CVS

showfilecontents.py

showfilecontents.py

exceptions > showfilecontents.py > ...

```
1  # Open file that's in this same folder.
2  thefile = open('people.csv')
3  # Show the filename.
4  print(thefile.name)
5
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) PS C:\Users\Alan\Desktop\exceptions> & C:/Users/Alan/anaconda3/python.exe c:/Users/Alan/Desktop/exceptions/showfilecontents.py
Traceback (most recent call last):
  File "c:/Users/Alan/Desktop/exceptions/showfilecontents.py", line 2, in <module>
    thefile = open('people.csv')
FileNotFoundError: [Errno 2] No such file or directory: 'people.csv'
(base) PS C:\Users\Alan\Desktop\exceptions>
```

FIGURE 7-3: The show `filecontents.py` file raises an exception.

`Traceback` is a reference to the fact that if there were multiple exceptions, they'd all be listed, with the most recent listed first. In this case, there is just one exception. The `File` part tells you where the exception occurred, in line `2` of the `showfilecontents.py` file. The following part shows you the line of code that caused the error:

```
thefile = open('people.csv')
```

And finally, the exception itself is described:

```
FileNotFoundError: [Errno 2] No such file or directory: 'people.csv'
```

The generic name for this type of error is `FileNotFoundError`. Many exceptions are also associated with a number (`ERRNO 2` in this example). But the number can vary depending on the operating system environment, so it's typically not used for handling errors. In this case, the main error is `FileNotFoundError`, and the fact that's its `ERRNO 2` where we're sitting right now doesn't matter.

Some people use the phrase *throw an exception* rather than *raise an exception*. The two phrases mean the same thing.

The last part tells you *exactly* what went wrong: `No such file or directory: 'people.csv.'` In other words, Python can't do the `open('people.csv')` business because there is no file named `people.csv` in the current folder.

You could correct this problem by changing the code, but `.csv` is a common file extension for files that contain comma-separated values. It would make more sense to change the name of `people.cvs` to `people.csv` so it matches what the program is looking for and the `.csv` extension is well known.

Handling Errors Gracefully

The best way to handle a file not found error is to replace what Python normally displays with something the person using the app is more likely to understand. To do that, you can code a *try . . . except block* using this basic syntax:

```
try:
     The things you want the code to do
except Exception:
     What to do if it can't do what you want it to do
```

Here's how you can rewrite the `showfilecontents.py` code to handle a missing (or misspelled) file error:

```
try:
    # Open file and show its name.
    thefile = open('people.csv')
    print(thefile.name)
except Exception:
    print("Sorry, I don't see a file named people.csv here")
```

Because the file that the app is supposed to open may be missing, we start with `try:` and then attempt to open the file under that. If the file opens, the `print()` statement runs and displays the filename. But if trying to open the file raises an exception, the program doesn't bomb and display a generic error message. Instead, it displays a message that the average computer user can understand, as shown in Figure 7-4.

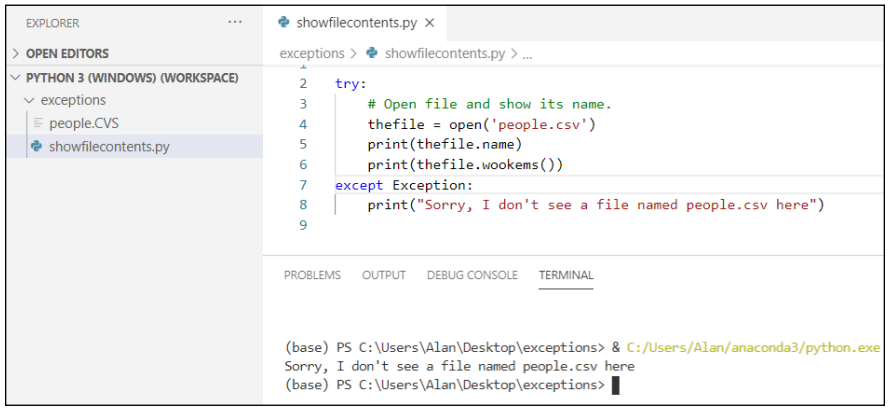


FIGURE 7-4: The show `filecontents.py` file catches the error and displays a friendly message.

Being Specific about Exceptions

Our previous code example handled the file not found error gracefully. But a larger app might have many places where there's a potential for error, and you want to handle each error differently. To accomplish this, you can define multiple error handlers, as we discuss next.

Suppose that you manually fix the filename so that it's `people.csv` as originally intended. As you saw, when you run the code and there's no error, the output is

just the filename. Below the line that prints the filename, we've added another line of code:

```
try:
    # Open file and show its name.
    thefile = open('people.csv')
    print(thefile.name)
    print(thefile.wookems())
except Exception:
    print("Sorry, I don't see a file named people.csv here")
```

When you run this code, the filename isn't a problem, so the output displays `people.csv`, as you'd expect. However, the next line of code, `print(thefile.wookems())`, throws an error because we haven't defined a method named `wookems()`. Unfortunately, the error message is still the same as it was before, even though the cause of the error is that there is no method in Python named `.wookems()`:

```
people.csv
Sorry, I don't see a file named people.csv here
```

So why is the error message saying that the file named `people.csv` wasn't found, when we know it *was* found and that the next line of code is causing the error? The problem is in the `except Exception:` line, which says "if *any* exception is raised in this `try` block, do the code under the `except` line."

To clean this up, you need to replace `Exception:` with the specific exception you want Python to catch. But how do you know what that specific exception is? Easy. The exception raised with no exception handing is

```
FileNotFoundError: [Errno 2] No such file or directory: 'people.csv'
```

The first word is the name of the exception that you can use in place of the generic `Exception` name, like this:

```
try:
    # Open file and show its name.
    thefile = open('people.csv')
    print(thefile.name)
    print(thefile.wookems())
except FileNotFoundError:
    print("Sorry, I don't see a file named people.csv here")
```

Granted, isolating the file not found error doesn't do anything to help with the bad method name. However, the bad method name isn't an exception; it's a programming error that needs to be corrected in the code by replacing `.wookems()`

with the method name you want to use. At least the error message you see isn't the misleading `Sorry, I don't see a file named people.csv here` error. The code works normally and therefore displays the filename when instructed. Then when it reaches the line that contains the bad `.wookem()` method, it throws an error — but not an error related to the filename not being found. It displays the correct error message for this error, `object has no attribute 'wookems'`, as shown in Figure 7-5.

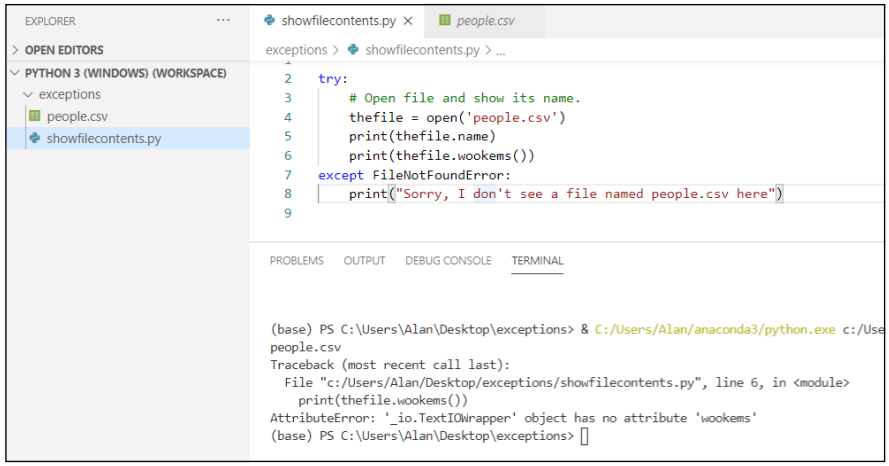


FIGURE 7-5: The correct error message is displayed.

Again, if you're thinking about handling the `.wookems` error, that's not an exception for which you'd write an exception handler. Exceptions occur when something *outside* the program upon which the program depends isn't available. Programming errors, such as nonexistent method names, are errors inside the program and have to be corrected there by the programmer who wrote the code.

Keeping Your App from Crashing

You can stack up `except:` statements in a `try` block to handle different errors. Just be aware that when the exception occurs, it looks at each one starting at the top. If it finds a handler that matches the exception, it raises that one. If some exception occurred that you didn't handle, you get the standard Python error message. But there's a way around that too.

If you want to avoid all Python error messages, you can start the last exception handler in the code with `except Exception:`. That line means "If the error that occurred wasn't already handled by one of the previous exceptions, use the

exception handler instead." In other words, the catch-all exception handler handles any exception that wasn't already handled in the code. For example, here we have two handlers, one for a file not found error and one for everything else:

```
try:
    # Open file and show its name.
    thefile = open('people.csv')
    # Print a couple blank lines then the first line from the file.
    print('\n\n', thefile.readline())
    # Close the file.
    thefile.closed()

except FileNotFoundError:
    print("Sorry, I don't see a file named people.csv here")
except Exception:
    print("Sorry, something else went wrong")
```


REMEMBER

We know that you haven't learned about `open` and `readline` and `close`, but don't worry about that. All we care about for now is the exception handling, which is the `try:` and `except:` portions of the code.

Running this code produces the following output:

```
Username,FirstName,LastName,Role,DateJoined

Sorry, something else went wrong
```

The first line displays the first line of text from the `people.csv` file. The second line is the output from the second `except:` statement, which reads `Sorry, something else went wrong`. This message is vague and doesn't help you find the problem.

Rather than just print a generic message for an unknown exception, you can capture the error message in a variable and then display the contents of that variable to see the message. As usual, you can name the variable anything you like, though a lot of people use `e` or `err` as an abbreviation for *error*.

For example, consider the following rewrite of the preceding code. The generic handler, `except Exception`, now has an `as e` at the end, which means "whatever exception gets caught here, put the error message in a variable named `e`." Then the next line uses `print(e)` to display the content of the `e` variable:

```
try:
    # Open file and show its name.
    thefile = open('people.csv')
    # Print a couple blank lines then the first line from the file.
```

```
    print('\n\n', thefile.readline())
    thefile.wigwam()

except FileNotFoundError:
    print("Sorry, I don't see a file named people.csv here")
except Exception as e:
    print(e)
```

Running this code displays the following:

```
Username,FirstName,LastName,Role,DateJoined

'_io.TextIOWrapper' object has no attribute 'wigwam'
```

The first line is just the first line of text from the `people.csv` file. There's no error in the code, and that file is there, so all went well. The second line is

```
'_io.TextIOWrapper' object has no attribute 'wigwam'
```

This isn't plain English, but it's better than "Something else went wrong." At least the part that reads `object has no attribute 'wigwam'` lets you know that the problem has something to do with the word *wigwam*. You handled the error gracefully and the app didn't crash. And you at least got some information about the error that should be helpful to you, even though it may not be helpful to people who are using the app with no knowledge of its inner workings.

Adding an else to the Mix

In our last working example, we used one error handler to handle file not found errors, and a second handler for everything else. But in real life, you may have to handle many more. And if there's no error, you want execution to continue normally. You can use the `else` for the last condition, as follows:

```
try:
     The thing that might cause an exception
catch a common exception:
     Explain the problem
catch Exception as e:
     Show the generic error message
else:
     Continue on here only if no exceptions raised
```

If you convert this code to plain English, the logic of the flow is as follows:

Try to open the file.

If the file isn't there, tell them and stop.

If there's some other error, show the generic error message and stop.

Otherwise

Go on with the rest of the code.

By limiting `try:` to the one thing that's most likely to raise an exception, we can stop the code dead in its tracks before it tries to go any further. But if no exception is raised, the code continues on normally, below the `else`, where the previous exception handlers don't matter anymore. Here is all the code with comments explaining what's going on:

```
try:
    # Open the file named people.csv
    thefile = open('people.csv')
# Watch for common error and stop program if it happens.
except FileNotFoundError:
    print("Sorry, I don't see a file named people.csv here")
# Catch any unexpected error and stop the program if one happens.
except Exception as err:
    print(err)
# Otherwise, if nothing bad has happened by now, just keep going.
else:
    # File must be open by now if we got here.
    print('\n') # Print a blank line.
    # Print each line from the file.
    for one_line in thefile:
        print(one_line)
    thefile.close()
    print("Success!")
```

WARNING

As always with Python, indentations matter a lot. Make sure you indent your own code as shown in this chapter. Otherwise, your code will not work right.

Figure 7-6 also shows all the code and the results of running that code in VS Code.

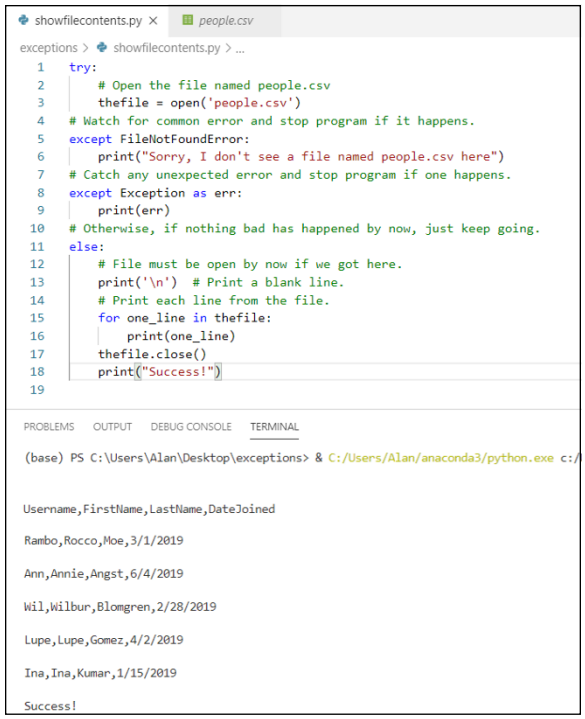


FIGURE 7-6: Code with try, exception handlers, and an else for when there are no exceptions.

Using try . . . except . . . else . . . finally

If you look at the complete syntax for Python exception handling, you'll see one more option at the end, like this:

```
try:
    try to do this
except:
    if x happens, stop here
except Exception as e:
    if something else bad happens, stop here
else:
    if no exceptions, continue on normally here
finally:
    do this code no matter what happened above
```

The `finally` code is executed when the `try` block ends *no matter what*. For example, if you're inside a function and an `except` block uses `return` to exit the function, the `finally` code *still* executes. Without that kind of feature, the `finally` block would be the equivalent of putting its code after and outside the `try` block.

To illustrate, here is some code that expects an external resource named `people.csv` to be available to the code:

```
print('Do this first')
try:
    open('people.csv')
except FileNotFoundError:
    print('Cannot find file named people.csv')
except Exception as e:
    print(e)
else:
    print('Show this if there is no exception.')
finally:
    print('This is in the finally block')
print("This is outside the try...except...else...finally")
```

When you run this code with a file named `people.csv` in the folder, you get this output:

```
Do this first
Show this if no exception.
This is in the finally block
This is outside the try...except...else...finally
```

None of the exception-reporting code executed because the `open()` statement was able to open the file named `people.csv`.

If you run this code without a file named `people.csv` in the same folder, you get the following result:

```
Do this first
Cannot find file named people.csv
This is in the finally block
This is outside the try...except...else...finally
```

This time the code reports that it can't find a file named `people.csv`. But the app doesn't crash. Rather, it keeps executing the rest of the code.

These examples illustrate that you can control exactly what happens in a small part of a program vulnerable to user errors or other outside exceptions while allowing other code to run normally.

Raising Your Own Exceptions

Python has lots of built-in exceptions for recognizing and identifying errors, as you'll see while writing and testing code, especially when you're first learning. However, you aren't limited to the built-in exceptions. If your app has a vulnerability that isn't covered by the built-in exceptions, you can invent your own.

For a detailed list of all the different exceptions that Python can catch, look at `https://docs.python.org/3/library/exceptions.html` in the Python.org documentation.

The general syntax for raising your own error is

```
raise error
```

Replace *error* with the name of the known error that you want to raise (such as `FileNotFoundError`). Or, if the error isn't covered by one of the built-in errors, you can just use `raise Exception` and that will execute whatever is under `catch Exception:` in your code.

As a working example, let's say you want two conditions to be met for the program to run successfully:

- The `people.csv` file must exist so you can open it.
- The `people.csv` file must contain more than one row of data. The first row contains column names, not data, so if the file has only column headings, we will consider it empty.

Here is an example of how you might handle the exception-handling part of that situation:

```
try:
    # Open the file ]
    thefile = open('people.csv')
    # Count the number of lines in file.
    line_count = len(thefile.readlines())
    # If there are fewer than 2 lines, raise exception.
    if line_count < 2:
        raise Exception
# Handles missing file error.
except FileNotFoundError:
    print('\nThere is no people.csv file here')
```

```
# Handles all other exceptions
except Exception as e:
# Show the error.
    print('\n\nFailed: The error was ' + str(e))
    # Close the file.
    thefile.close()
```

So let's step through the code. The first lines try to open the `people.csv` file:

```
try:
    # Open the file (no error check for this example).
    thefile = open('people.csv')
```

We know that if the `people.csv` file doesn't exist, execution will jump to the following exception handler, which tells the user the file isn't there:

```
except FileNotFoundError:
    print('\nThere is no people.csv file here')
```

Assuming the file was found and no error was thrown, and the file is now open, this next line counts how many lines are in the file:

```
line_count = len(thefile.readlines())
```

If the file is empty, the line count will be `0`. If the file contains only column headings, like this:

```
Username,FirstName,LastName,DateJoined
```

the length will be `1`. We want the rest of the code to run only if the length of the file is `2` or more. So if the line count is less than `2`, the code will raise an exception. You may not know what that exception is, so you tell the app to raise a general exception with `raise Exception` (with an uppercase *E*):

```
if line_count < 2:
    raise Exception
```

The exception handler for general exceptions looks like this:

```
# Handles all other exceptions
except Exception as e:
    # Show the error.
    print('\n\nFailed: The error was ' + str(e))
    # Close the file.
    thefile.close()
```

The `e` variable grabs the exception, and the next `print` statement displays the exception. So, let's say you run that code and `people.csv` is empty or incomplete. The output would be

```
Failed: The error was
```

Note that there is no explanation of the error because we're using

```
except Exception as e:
```

Remember that `Exception` refers to any error, not an error that has a specific name stored in the variable named `e`. To throw an error that has an error message associated with it, replace `Exception` with a specific Python exception name. For example, in the following code we've replaced the generic `Exception` with the more specific `FileNotFoundError`:

```
if line_count < 2:
    raise FileNotFoundError
```

But if you do that, the `FileNotFoundError` handler is called and displays `There is no people.csv file`, which isn't true in this case and it's not the cause of the problem. There is a `people.csv` file; it just doesn't have any data to loop through. What you need is a custom exception handler for that exception.

All exceptions in Python are objects, instances of classes that inherit from the base class `Errors` in Python. To create your own exception, you first have to import the `Exception` class to use as a base class (much like the `Member` class was a base class for different types of users). Then you define your error as a subclass of that base class. This code goes at the top of the file so it's executed before any other code tries to use the custom exception:

```
# Define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass

# Your custom error (inherits from Error)
class EmptyFileError(Error):
    pass
```

As before, the word `pass` in each class tells Python "I know this class has no code in it, and that's okay here. You don't need to raise an exception to tell me that."