LEARNING MADE EASY

Python®

ALL-IN-ONE

for dummies®
A Wiley Brand

7 Books in one!

John Shovic
Alan Simpson

Python

ALL-IN-ONE

by John Shovic and Alan Simpson

for dummies®
A Wiley Brand

Python All-in-One For Dummies®

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Published simultaneously in Canada

Library of Congress Control Number: 2019937504

ISBN 978-1-119-55759-3 (pbk); ISBN 978-1-119-55767-8 (ebk); ISBN 978-1-119-55761-6 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents at a Glance

Table of Contents

Introduction

The power of Python. The Python language is becoming more and more popular, and in 2017 it became the most popular language in the world according to IEEE Spectrum. The power of Python is real.

Why is Python the number one language? Because it is incredibly easy to learn and use. Part of it is its simplified syntax and its natural-language flow, but a lot of it has to do with the amazing user community and the breadth of applications available.

About This Book

This book is a reference manual to guide you through the process of learning Python and how to use it in modern computer applications, such as data science, artificial intelligence, physical computing, and robotics. If you are looking to learn a little about a lot of exciting things, then this is the book for you. It gives you an introduction to the topics that you will need to go deeper into any of these areas of technology.

This book guides you through the Python language and then it takes you on a tour through some really cool libraries and technologies (the Raspberry Pi, robotics, AI, data science, and so on) all revolving around the Python language. When you work on new projects and new technologies, Python is there for you with an incredibly diverse number of libraries just waiting for you to use.

This is a hands-on book. There are examples and code all throughout the book. You are expected to take the code, run it, and then modify it to do what you want. You don't just buy a robot, you build it so you can understand all the pieces and can make sense of the way Python works with the robot to control all the motors and sensors. Artificial intelligence is complicated, but Python helps make a significant part of it accessible. Data science is complicated, but Python helps you do data science more easily. Robotics is complicated, but Python gives you the code that controls the robot. And Python even allows us to tie these pieces together and use, say, AI in robotics.

In this book, we take you through the basics of the Python language in small, easy-to-understand steps. After we have introduced you to the language, then we

step into the world of Python and artificial intelligence, exploring programming in machine learning and neural networks using Python and TensorFlow and actually working on real problems and real software, not just toy applications.

After that, we're off to the exciting world of Big Data and data science with Python. We look at big public data sets such as medical and environmental data all using Python.

Finally, you get to experience the magic of what I call "physical computing." Using the small, inexpensive Raspberry Pi computer (it's small, but incredibly popular) we show you how to use Python to control motors and read sensors. This is a lead-up to our final book, "Python and Robotics." Here you learn how to build a robot and how to control that robot with Python and your own programs, even using artificial intelligence.

This is not your mother's RC car.

Python data science, robotics, AI, and fun all in the same book.

This book won't make you understand everything about these fields, but it will give you a great introduction to the terminology and the power of Python in all these fields. Enjoy the book and go forth and learn more afterwards.

Foolish Assumptions

We assume you know how to use a computer in a very basic way. If you can turn on the computer and use a mouse, you're ready for this book. We assume you don't know how to program yet, although you will have some skills in programming by the end of the book. If we're wrong and you do already know Python (or some other computer language), jump ahead to minibook 4 and dig right into learning something new. Our intent is to guide you through the language of Python and then through some of the amazing technologies and devices that use Python. We provide complete examples. If you get stuck on something, look it up on the web, read a tutorial, and then come back to it.

Icons Used in This Book

What's a *For Dummies* book without icons pointing you in the direction of truly helpful information that's sure to speed you along your way? Here we briefly describe each icon we use in this book.

TIP

The Tip icon points out helpful information that's likely to make your job easier.

REMEMBER

This icon marks a generally interesting and useful fact — something you may want to remember for later use.

WARNING

The Warning icon highlights lurking danger. When we use this icon, we're telling you to pay attention and proceed with caution.

TECHNICAL STUFF

When you see this icon, you know that there's techie-type material nearby. If you're not feeling technical-minded, you can skip this information.

Beyond the Book

In addition to the material in the print or ebook you're reading right now, this product also comes with some access-anywhere goodies on the web. No matter how well you understand Python concepts, you'll likely come across a few questions where you don't have a clue. To get this material, simply go to `www.dummies.com` and search for "*Python All-in-One For Dummies* Cheat Sheet" in the Search box.

Where to Go from Here

Python All-in-One For Dummies is designed so that you can read a chapter or section out of order, depending on what subjects you're most interested in. Where you go from here is entirely up to you!

Book 1 is a great place to start reading if you've never used Python before. Discovering the basics and common terminology can be quite helpful for later chapters that use the terms and commands regularly!

Occasionally, we have updates to our technology books. If this book does have any technical updates, they'll be posted at `www.dummies.com/go/pythonaiofdupdates`.

1 Getting Started with Python

Contents at a Glance

IN THE CHAPTER

- Why Python is hot
- Tools for success
- Writing Python in VS Code
- Writing Python in Jupyter notebooks

Chapter 1
Starting with Python

The fact that you're reading this implies you know that Python is a great thing to know if you're looking for a good job in programming. It's also good to know if you're looking to expand your existing programming skills into exciting cutting-edge technologies like 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, especially in this book, leans heavily toward the hands-on. A common failure in many tutorials is that they already assume you're a professional programmer in Python or some other language, and they skip over things they assume you already know.

This book is different in that we *don't* assume you're already programming in Python or some other language. We assume you can use a computer and understand basics like files and folders. But that's about it for assumptions.

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 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. Personally, we've never seen anybody 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 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. Python is hot — that's probably why you want to learn it, and that's good. 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 really driving the whole 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 those areas, while many older languages lag behind in these technologies.

If you're not familiar with programming languages like C and Java, feel free to skip to the next section, "Python versions," as this information is only for people who wonder about differences among the languages. But in case you're wondering, 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 like 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) Python is free.
- Python offers more readymade tools for current hot technologies like data science, machine learning, artificial intelligence, and robotics than most other languages.

HTML, CSS, AND JavaScript

Some of you may have heard of languages like 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. JavaScript is a programming language; however, it too is heavily geared to Website development and isn't quite in the same category of general programming languages like Python and Java.

Another way to look at it is, you wouldn't learn HTML, CSS, and JavaScript instead of Python, as there is too little overlap to justify it. 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.

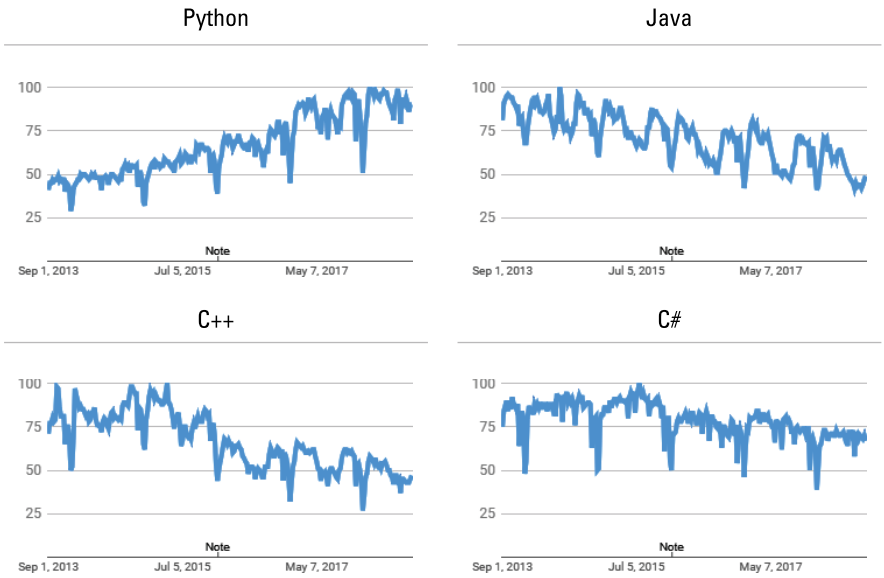


FIGURE 1-1: Google search trends for the last five years or so.

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 like

- 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 or a 1990 Ford Mustang or a 2000 Ford Mustangs, and a 2019 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 that are 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 us the newest, most recent "model," sporting improvements based on experience with earlier versions, as well as features that are 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 it's changed." When there's a significant change, the whole 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 here because there is little reason to know or understand the differences between all the versions. We only present the table 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 3.7	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	September 2012
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 1998
Python 1.0	January 1994

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

The car years analogy is just an analogy indicating that the larger the number, the more recent the version. But in Python it's 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, they will tell you what the most current stable build (version) is. 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 that company requires you to learn and use a specific version. That sort of thing 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 there are companies heavily invested in some 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 2018 and early 2019, from Python 3.7 and above. Don't worry about version differences after the first and second digits. Version 3.7.2 is similar enough go version 3.7.1 that it's not important, especially to a beginner. Likewise, Version 3.8 isn't that big a jump from 3.7. So don't worry about these miner version differences when first learning. Most of what's in Python is the across all versions. So you need not worry about investing time in learning a version that's obsolete or soon will be — unless you happen to be learning from a very old book.

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, 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 or writing code. A code editor is an app that lets you type code, in much the same way an app like Word or Pages helps you type regular plain-English text.

Just as there are many brands of toothpaste, soap, and shampoo in the worlds, there are many "brands" of code editors that work well with Python. There isn't a right one or wrong one, a good one or bad one, a best one or 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 thing is "good."

If you already started learning Python on your own before this book, 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 is . . .

An excellent, free learning environment

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

- It is an excellent editor for learning coding.
- It is an excellent editor for writing code professionally, and is in fact used by millions or 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 doing Python code. But you also need the Python interpreter. Chances are, you're also going to want some Python packages, too. The packages are simply code already 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 they are used 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 certainly 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 actually learning Python itself.

An excellent alternative to the old command-line driven ways of doing thing is to use a more complete Python development environment with a more intuitive and more easily managed graphic user interface, as on a Mac or Windows or any phone or tablet. The one we recommend is called Anaconda. It is free, and it is 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 `https://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 really take you step-by-step through every part of downloading and installing Anaconda because it's distributed from the a 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. You just have to keep an eye on your screen as you go along, and follow any onscreen instructions as they arise, while following the steps.

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 in Google 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 `https://www.anaconda.com/download/` to get to a page that looks something like this (don't worry about version numbers or dates, just download whatever they recommend when you get there).**
2. **Scroll down a little and you should see some download options that look something like the example shown in Figure 1-2.**

   We used a Windows computer for that screenshot, but Mac and Linux users will see something similar.

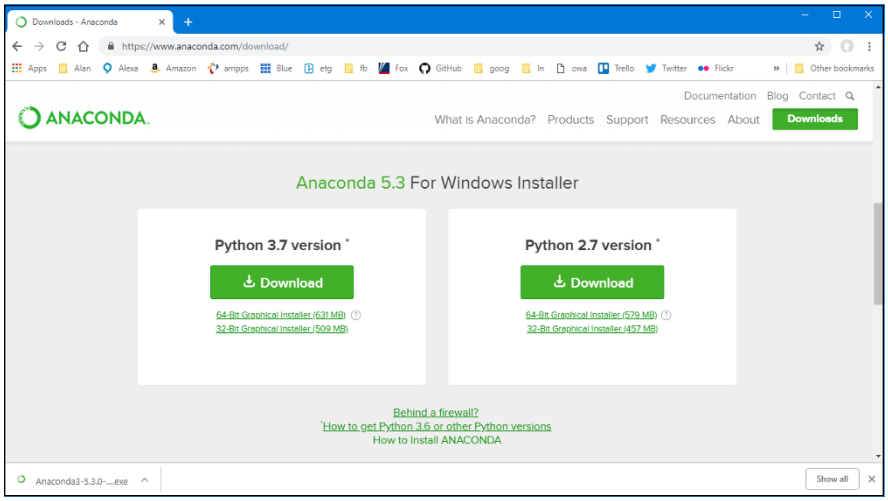


FIGURE 1-2: Click Download under the largest version number.

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

   For me, right now, it's version 3.7 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 on the download page if you'd like to see the instructions from the Anaconda team.

4. **If prompted for your email address, either provide it or click the X in the pop-up window's upper-right corner to close the prompt without entering your email address.**

5. **If you see Keep/Discard options in the lower-left corner of your screen, click Keep.**

6. **When the download is complete, open your Downloads folder (or whichever folder to which you downloaded the file).**

7. **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.

8. **Click Next, Continue, Agree, or I Agree on the first installation pages until you get to one of the pages shown in Figures 1-4 (Mac will be on the one on the left, Windows the one on the right).**

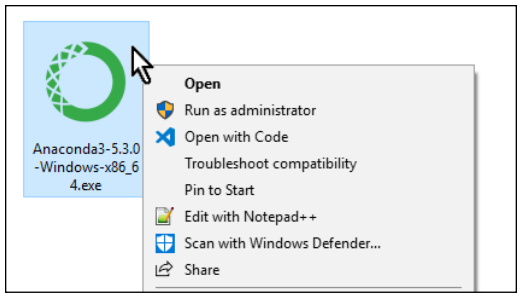


FIGURE 1-3: In Windows, right-click and choose Run As Administrator.

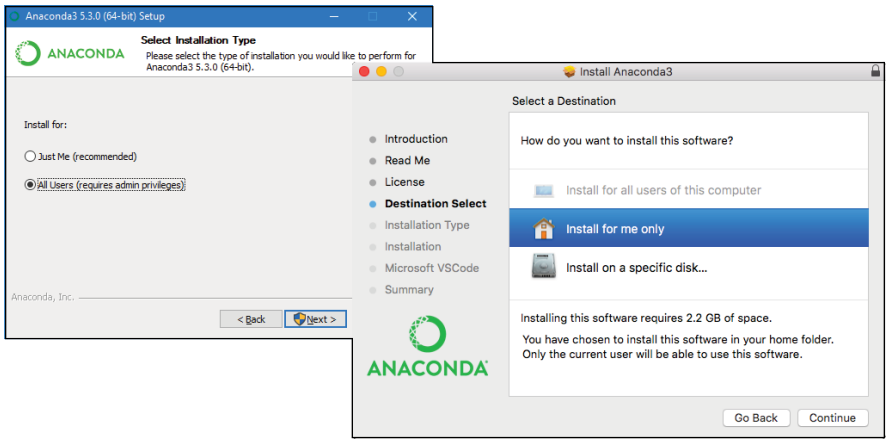


FIGURE 1-4: Choose how to install Anaconda.

9. **Choose whichever option makes sense to you. Or, if in double, choose Install on a specific disk (for Mac), and then Macintosh HD, or choose Install for All Users (if on Windows using Administrator privileges. If the option we suggested isn't available to you, click the one closest to it.**

10. **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. Just accept the default suggestions.**

11. **It may take several minutes, but eventually you'll come to a page where it asks if you want to installed Microsoft VS Code. 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.

12. **Continue to follow any onscreen instructions, click Continue or Next to proceed through the installation steps, 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. So you can decide for yourself if that's something you want to do.

Opening Anaconda (Mac)

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

- Open Launchpad and open Anaconda Navigator.
- Or click the Spotlight magnifying glass, start typing **Anaconda**, and then double-click Anaconda Navigator.
- Or open Finder and your Applications folder and double-click the Anaconda-Navigator icon there.

After Anaconda Navigator opens, right-click its icon in the Dock and choose Keep in Dock to keep its icon visible in the Dock at all times so it's easy to find when you need it.

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 just about any of these options:

- Click the Start button then click Anaconda Navigator on the Start menu.
- Or click the Start button, start typing **Anaconda**, then click Anaconda-Navigator on the Start menu once 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 it, it open to the Anaconda Navigator home page, which should look something like Figure 1-5.


TIP

If you see a prompt for getting 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.

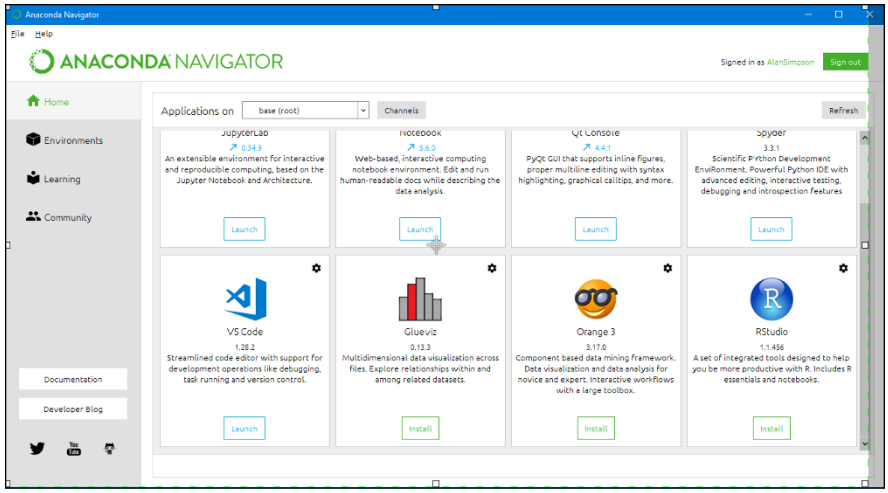


FIGURE 1-5: Anaconda Navigator Home page.

Down the left side of the Anaconda Navigator home page you see options like Home, Environments, Learning, Community, Documentation, and Developer Blog. You're welcome to explore these on your own. However, they're not directly related to learning and doing Python, so we'll let you choose which of those, if any, you're interested in exploring.

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" to version of Python that comes with Anaconda, which is easier than trying to figure out all of that yourself. So the steps are

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

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

ABOUT GIT

Git is a popular means of storing backups of your coding projects, and sharing 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 entirely 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 in the future if you later decide to learn about it.

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.

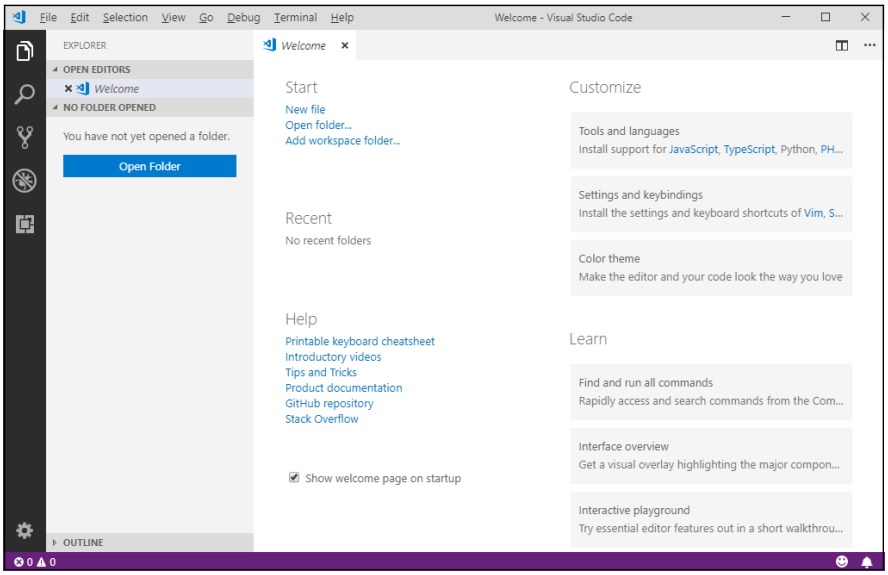


FIGURE 1-6: VS Code editor with welcome screen.

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, on a Mac click Code on the menu and choose Preferences➪Color Theme. In Windows, choose File➪Preferences➪Color Theme. Then choose a lighter color theme, like Light (Visual Studio) and your VS Code screens will look more like the ones in this book.

To make sure you're ready to do Python coding, click the Extensions icons 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.

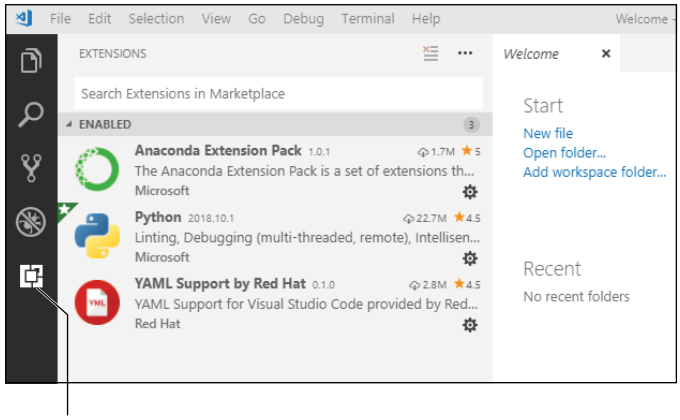


FIGURE 1-7: VS Code extensions for Python.

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. Do to 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.

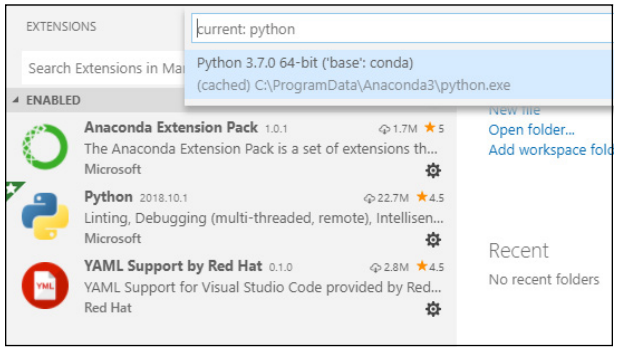


FIGURE 1-8: Choose your Python interpreter (usually the highest version number).

Writing some Python code

To test everything to make sure it's going to work, 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.

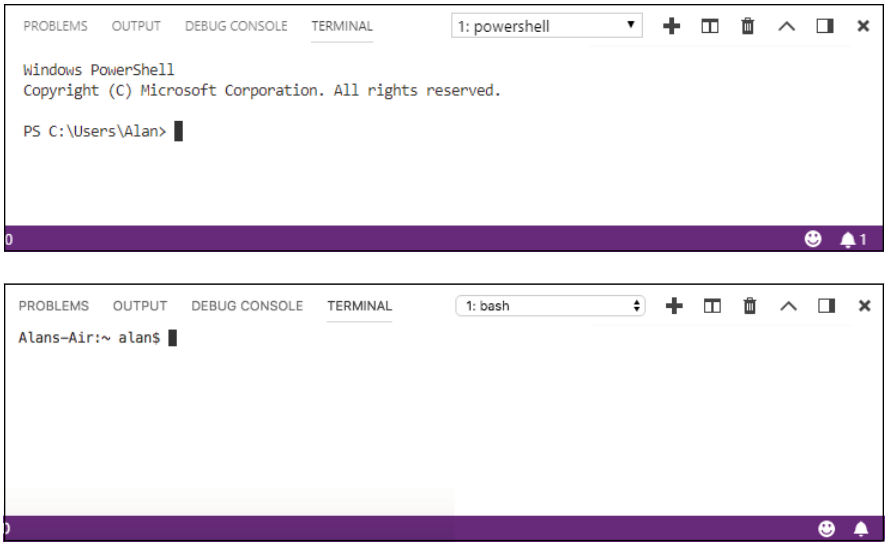


FIGURE 1-9: Terminal in VS Code (Windows and Mac).

2. **In the 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 one plus one), followed by another Python prompt, as shown in Figure 1-10.

```
(base) C:\Users\Alan\OneDrive\AIO Python>python
Python 3.7.0 (default, Jun 28 2018, 08:04:48) [MSC v.1912 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>>
```

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

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 the 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. Now let me show you how to exit out of Python and VS Code. Here are the steps:

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 in Windows, click the Close (X) button in the upper-right corner or choose View➪Exit from the menu. On a Mac, click the round red dot in the upper-left corner, or choose Code➪Quit Visual Studio Code from the menu.**
3. **You can also close Anaconda Navigator using similar techniques: click the X in the upper-right corner or choose File➪Quit from the menu bar in Windows. Or 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 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:

Julia

Python

R

Julia and R are popular for data science, Python is of course, 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.

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

1. **Open Anaconda as discussed earlier in this chapter**
2. **Click Launch under Jupyter Notebook (shown in Figure 1-11).**

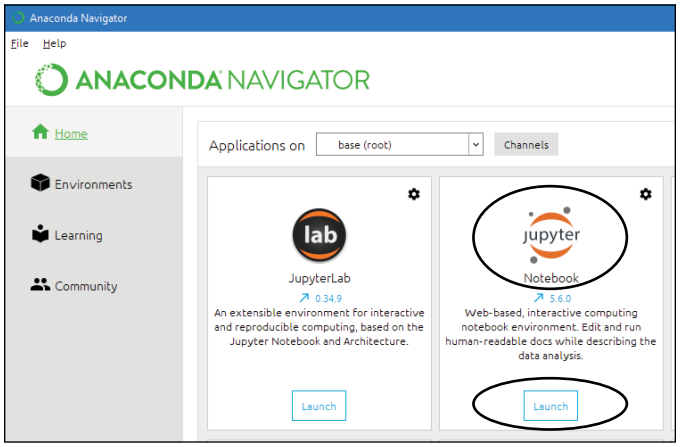


FIGURE 1-11: Launch Jupyter Notebook from Anaconda's home page.

Jupyter notebooks are web-based, meaning that when Jupyter opens, it does so in your default Web browser, which may be Safari, Google Chrome, Edge, 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 different from those in the figure, because those folder names are from my computer, not yours.)

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 [ ]:` at 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 shows the word Code. You can change that to Markdown if you want to write regular text rather than Python code.

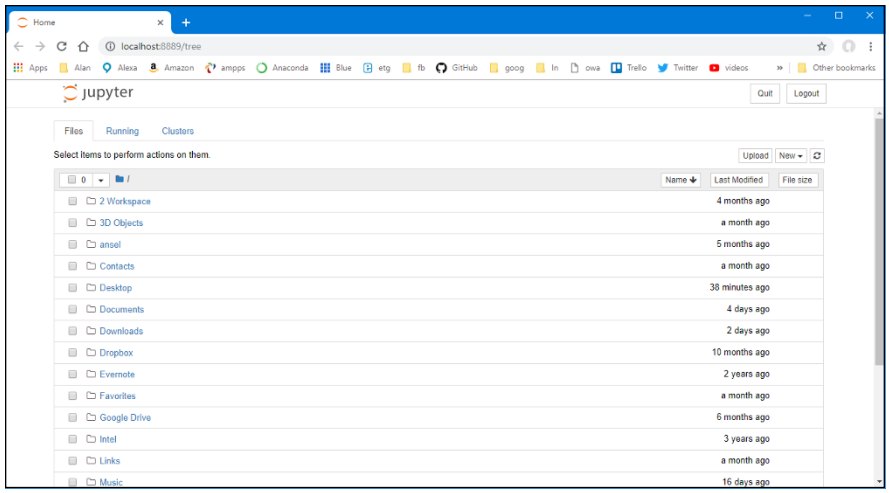


FIGURE 1-12: Jupyter Notebook opening page.

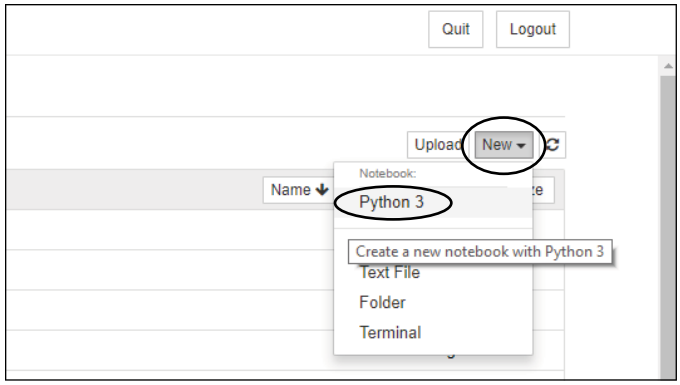


FIGURE 1-13: Creating a new Jupyter notebook.

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 just stay focused on Python code, since that 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 using 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 the 1+1 in the cell, but not result, 2. To get the result, click Run in the toolbar or put the mouse pointer into the cell and click the Run icon at the left side of the cell, as shown in Figure 1-14.

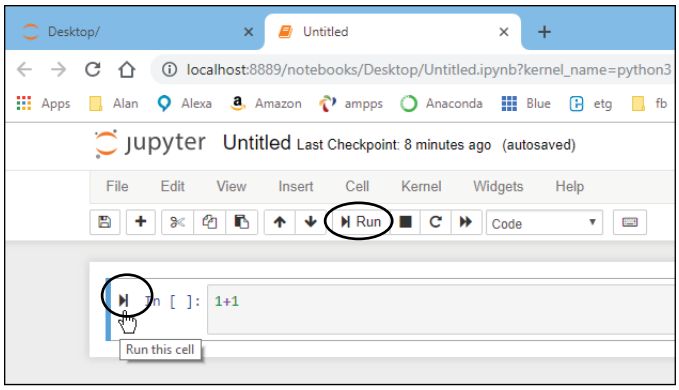


FIGURE 1-14: Two ways to run code in a Jupyter cell.

You'll see the number 2 to the right of Out[1], as in Figure 1-15. The Out indicates that you're seeing the output from executing the code in the cell, which of course is 2 because one plus one is two.

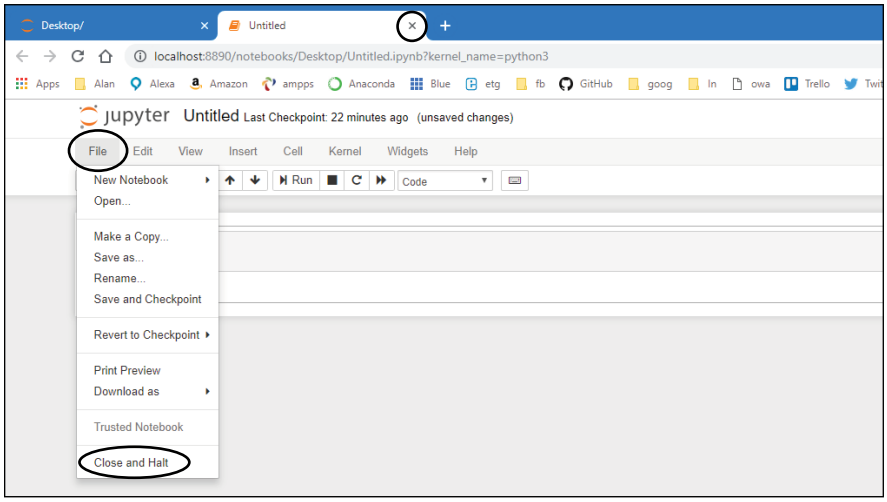


FIGURE 1-15: Result of running code in a Jupyter Notebook cell.

To close a notebook, you can used either of these methods:

- Close the tab in the browser that's showing the cell.
- Or, 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. You may be prompted to save your work. For now, you don't need to save because we're just focused on the absolute

basics . . . the thing you may be doing every time you want to run some Python code.

Even if you don't specifically save a notebook, you will see an icon for it in the folder in which you created the notebook. Its name will be Untitled, and if you have filename extensions visible, you'll see a .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 are just practicing and don't want to keep it. Just make sure you close the notebook in the web browser (or just close the whole browser first) — otherwise, you may get an error message stating that you can't delete the file while it's open.

So now you are ready to go. You have 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 you professional programming after you've mastered the basics. Come on over to Chapter 2 in this minibook now and we'll get a bit deeper into Python and using the tools you now have available on your computer.

IN THIS CHAPTER

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

Chapter 2

Interactive Mode, Getting Help, Writing Apps

Now that you have Anaconda and VS Code installed, 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 like data science, artificial intelligence, robotics, or whatever. But learning that will be easier if you have a good understand of the many tools available to you, and the skills to use them.

Using Python Interactive Mode

Many teachers and authors will suggest 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 steps from the fact that they're typing and executing the code in the wrong place. With Anaconda, the Terminal in VS Code is a great place to type Python code. So in this chapter we'll start with that.

Opening Terminal

To use Python interactively with Anaconda, we suggest you follow these steps:

1. **Open Anaconda Navigator, 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 words Terminal isn't highlighted at the top of the pane, click Terminal (circled in Figure 2-1).**

FIGURE 2-1: Terminal pane in VS Code.

The very first prompt you see is typically for your computer's operating system, and likely shows the user name of the account you're in. 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 path that's different from `C:\Users`.

Getting your Python version

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

```
python --version
```

COLORS AND ICONS IN VS CODE

By default, the VS Code terminal displays white text against a black background. We will be reversing those colors in this book, just because we think the dark against light works better for print like this. You're welcome to use any color scheme you like. If you just want to switch to black on white, as shown in this book, use either of these methods:

On a Mac choose Code ➪Preferences ➪Color Theme ➪Light (Visual Studio).

In Windows choose File ➪ Preferences ➪Color Theme ➪Light (Visual Studio).

If you want your icons in VS Code to match the ones we're using, 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 won't be using it for the book because it doesn't play well when printed on paper. But you may want to take it for a spin. Follow these steps:

1. **Click Extensions in the left pane.**
2. **Type** material**, look for Material Icon Theme, and click its Install option.**
3. **Click Reload on any selected extension to install both extensions. If you see a prompt at bottom right asking if you want to activate the icons, click Activate.**
4. **Choose File (in Windows) or Code (on a Mac) then Preferences ➪File Icon Theme then click Material Icon Theme.**
5. **If you'd like to try out the color theme, open File (in Windows) or Code (on a Mac) and then choose Preferences ➪Color Theme and click Material Icon Theme.**

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

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 within Anaconda, not just from Launchpad 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 know 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 will be waiting for a long, long time.

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 the 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 if you enter the command

```
howdy
```

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. Though in the past we've gotten different results with different VS Code versions. It's possible that the first time you try to use Python you'll see some messages in the lower-right corner of VS Code. Don't be alarmed if you don't see them. If you do see any, however, here is how you can respond:

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.

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

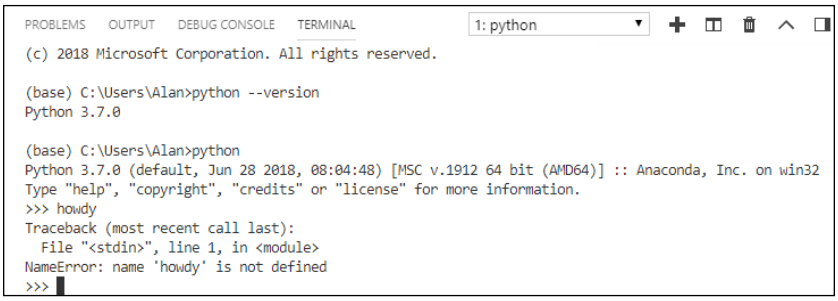


FIGURE 2-2: Python doesn't know what *howdy* means.

Using Python's built-in help

On of the prompts on your screen mentioned that you can type `help` as a comment 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.
```

Note that this time they're 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 they use the word "type" at the start of the sentence, they mean to enter the command. . .type it and press Enter. Go ahead and enter

```
help()
```

Note that there are no spaces in the line. After you press Enter the screen provides some information about using Python's interactive help, as shown in Figure 2-3.

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. Any commands you type here should be ones that the help recognizes. As a beginner you're not likely to know those. But for future reference know you can use this help prompt for reminders about different parts of Python.

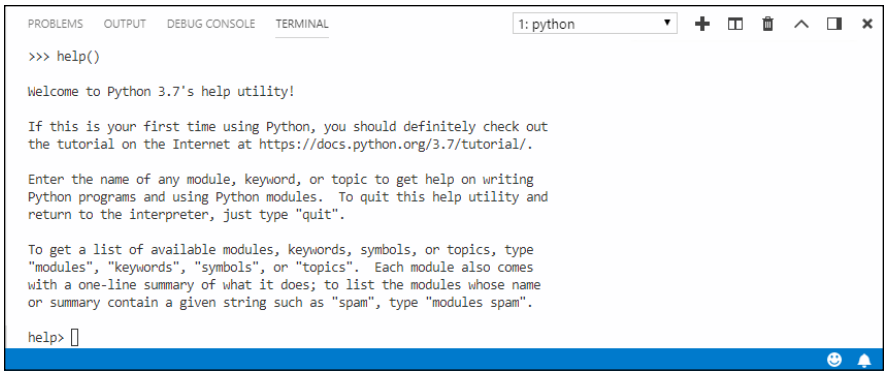


FIGURE 2-3: Python's interactive help utility.

For example, Python uses certain keywords, which have special meaning in the language. You get a list of those, just type

```
keywords
```

at the `help>` prompt and press Enter, and you'll see a list of keywords, as in Figure 2-4.

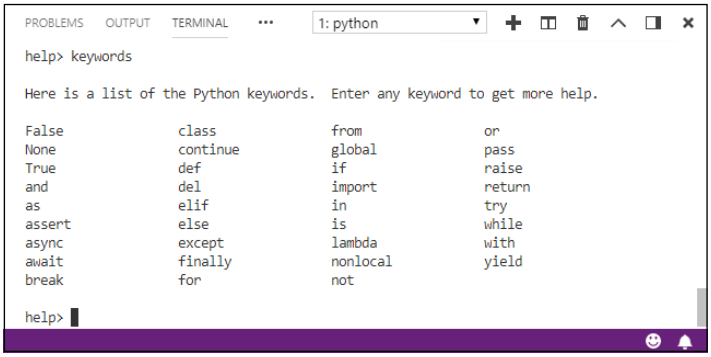


FIGURE 2-4: Keyword help.

Above the list of keywords it tells you that you can type any keyword at the `help>` prompt for more information about that keyword. For example, entering the keyword `class` 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).

Needless to say, all the technical jargon in the help text is going to leave the average beginner totally flummoxed. But the point here is that, for future reference, as you learn about things in Python, you can use the Python interactive help for reminders on those concepts as needed.

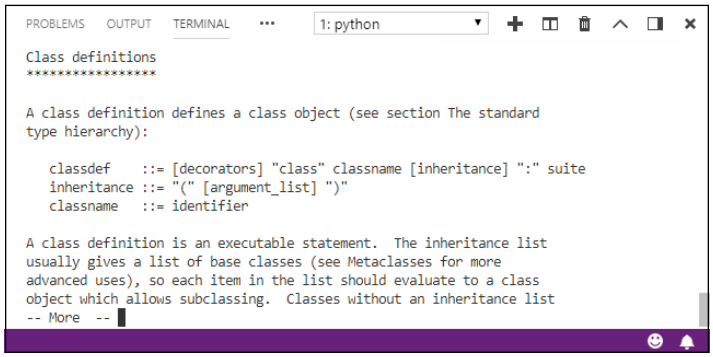


FIGURE 2-5:
Python class help.

The `--More--` at the bottom of the text isn't a prompt where you type commands. Instead, it's just letting you know that there is more text to come. Press the Spacebar. There may be several pages of information. Every time you see `-- More --`, you can press Enter to get to the next page. Eventually you'll get back to the `help>` prompt, and that's when you know you've reached the end of that help.

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, then press Enter. You should be back to the `>>>` prompt. At the `>>>` prompt you can 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 leaving off the parentheses, you'll get some help on the screen. For example, if you enter `exit` and press Enter you see

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

Thist tells you that in order to exit the Python prompt you should type **exit()** (with the parentheses, no spaces), or press Ctrl+Z, then press Enter. 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.

Searching for specific help topics online

Python's built-in help is somewhat archaic, but it can help you when you just need a quick reminder about some Python keyword you've forgotten. But if you're online, you're probably better off just searching the Web for help. You may want to start at `https://www.youtube.com/` if you're specifically looking for videos, and 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.

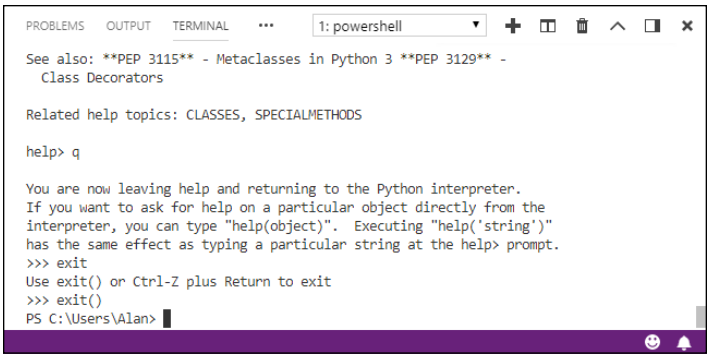


FIGURE 2-6: Back to the operating system prompt.

Regardless of what you use to search, remember to start your search with the word *python* or *python 3*. A lot of programming languages out there 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

Another good resource 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 down to a single page or so can really help bring things down to a more manageable (and less intimidating) size.

Of course, you're not really "cheating" with a cheat sheet, unless you us 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 really just another tool to help you learn. There are many of them out there, and exactly what appeals to you depends on your own learning style. To see what's available, just head out 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 and the rest are certainly decent support tools, most people want to use Python to create apps. Personally, we've found this easiest to do if you set up a VS Code development environment specifically for

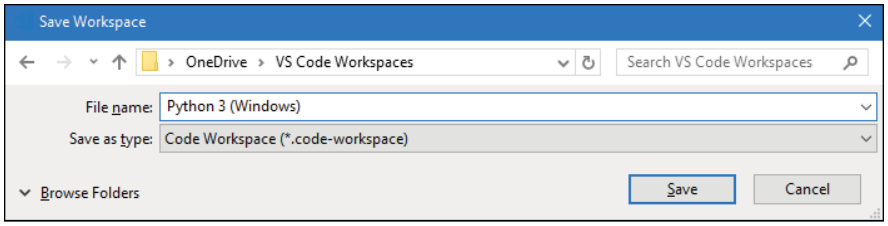
learning and doing Python. You can set up other development environments for other types of coding, such as HTML, CSS, and JavaScript for the Web, fine-tuning each as you go along to best support whatever language you're working in.

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

In VS Code, they use the term *workspace* to define what we call a development environment. It's basically the specific Python interpreter you're using plus any additional extensions you gather along to the way to make learning and doing easier.

To make it easy to get to these workspaces from any Internet-connected computer in the world, Alan has a folder on a cloud drive (OneDrive) named VS Code Workspaces. However, you can store your workspaces anyplace you like — even on your own computer, if you don't have or don't want to use a cloud drive. But if you do want to create a folder for storing workspaces, do so now, before proceeding with the following steps. Then

1. **If VS Code isn't already open, open launch it from Anaconda.**
2. **Choose File ➪ Save Workspace As.**
3. **Navigate to the folder in which you want to save the workspace settings.**
4. **Type a name for the workspace. Alan uses** Python 3 **followed by** Mac **or** Windows **depending on which type of machine he's on, as shown in Figure 2-7.**
   - On a Mac, choose Code ➪ Preferences ➪ Settings.
   - In Windows, choose File ➪ Preferences ➪ Settings.
5. **If you see a page like the one in Figure 2-7, click the three dots near the top-right corner and choose Open Settings.json, as shown in Figure 2-8.**
6. **In the next window, select the entire line of code that starts with** `python.pythonpath` **(this 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. **Click Workspace Settings above the code you just selected.**



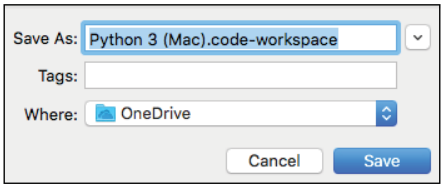


FIGURE 2-7: Saving current settings as workspace settings.

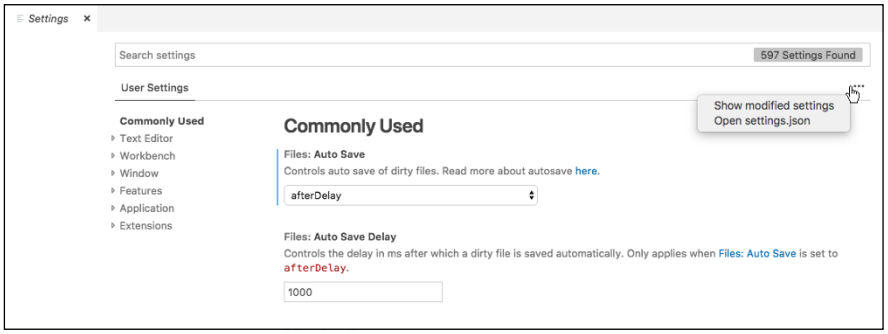


FIGURE 2-8: VS Code Settings.

8. Click between the setting's curly braces and paste in the lines of code there, as shown in Figure 2-9. Keep in mind that your python path may not look like the one in the image.

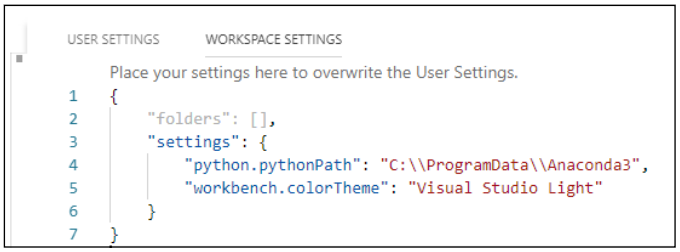


FIGURE 2-9: Python path moved to workspace settings.

9. **Choose File➪Save from the VS Code menu.**
10. **Close the Settings and User Settings tabs by clicking the X on the right side of each tab.**
11. **Close VS Code.**
12. **Close Anaconda.**

You'll see how to take advantage of the new workspace settings in a moment. But first, you'd be wise to create a directory (folder) to store all the code you'll be writing in this book so it's easy to find when you want to review.

Creating a Folder for your Python Code

Next, we create a folder in which 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. Alan again will use a cloud drive (OneDrive) so he can get to it from any computer. But you can put yours wherever you like. He'll name his folder *AIO Python* (for All-In-One Python), but you can name yours whatever you like.

The steps are the same as for any other folder, there's nothing special about the folder. In Windows you can navigate to the folder which will contain the new folder (Alan would use OneDrive, but you can use your Desktop, Documents, or any other folder). On a Mac, right-click some empty place in the folder and choose New Folder. In Windows, right-click an empty spot in the folder and choose New➪Folder. Type the folder name (*AIO Python*, in our example) and press Enter.

Last but not least, you want to associate the code folder you just created with the VS Code workspace you just created, so that any time you work in the AIO Python 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 AIO Python folder, click that folder's icon, and choose Add.**

The Explorer bar in VS Code shows that you have the workspace open, and under that you can see you also have that folder open, as shown in Figure 2-10. If you see something entirely different in the left pane, click the Explorer icon at the top-left (near the arrow in the Figure) to make sure you're viewing that pane.

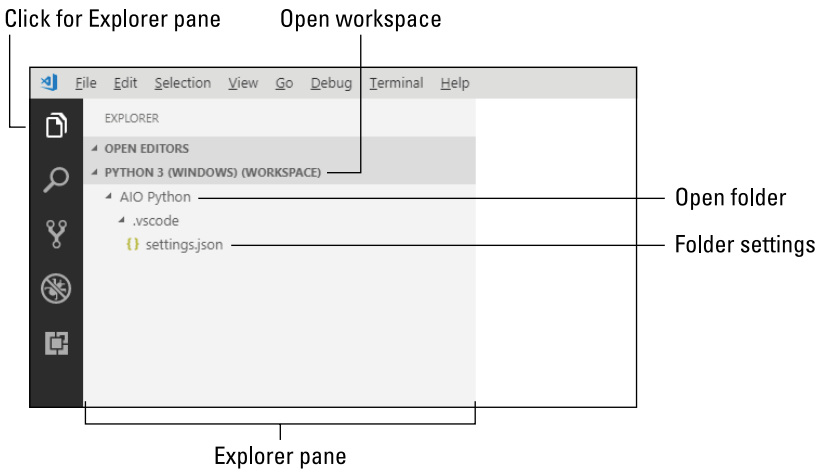


FIGURE 2-10: Python 3 workspace and AIO Python folder open in VS Code.

The Open Editors bar lists files that are currently open (over to the right). For example, if you see Welcome there, that means you're viewing the Welcome page. To close that page, click the X next to its name in the Explorer pane or tab. Any time you want to reopen that Welcome page, choose Help➪Welcome from the VS Code menu bar.

The .vscode icon is just a subfolder that stores your settings for this workspace, and settings.json is the file in that folder that contains those settings. The triangle next to each name allows you to expand or collapse that list. So if something is hidden, click the triangle next to an item to expand that item and see what it contains.

TIP

If you see a symbol other than a triangle, or no symbol at all, next to folder names, 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/collapse it even if there's no symbol there at all.

The beauty of this approach now 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 up, and you're ready to go.

Typing, Editing, and Debugging Python Code

Most likely, the vast majority of code you write you'll write in and editor. This will be a plain text file with a .py filename extension. For this book, we suggest you keep any files you create in that AIO Python folder which you should be able to see any time you have VS Code and your Python 3 workspace open. So to create a .py file at any time, follow these steps:

1. **If you haven't already done so, open VS Code on 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 new file in your AIO Python folder, right-click AIO Python and choose New File (Figure 2-11).**
4. **Type the filename with the .py extension (**hello.py **for this first one) and press ENTER.**

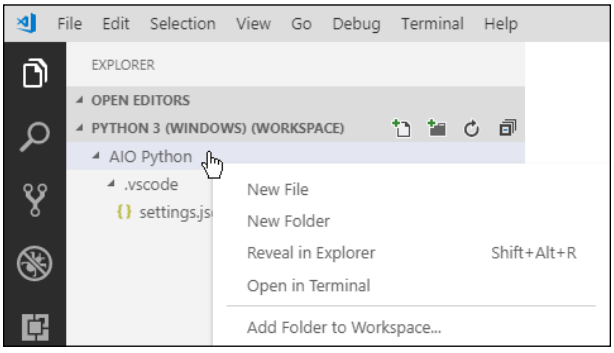


FIGURE 2-11: Right-click a folder name and choose New File.

The new file opens and you can see its name on top of the tab to the right. That larger area is where you'll type your Python code. You'll also see hello.py under Open Editors. That's just a convenience so that when you have many open files you can pick one to bring to the forefront just by clicking its name there. The filename also appears under the AUO Python folder name in the Explorer pane, because that's where it's stored, as shown in Figure 2-12.

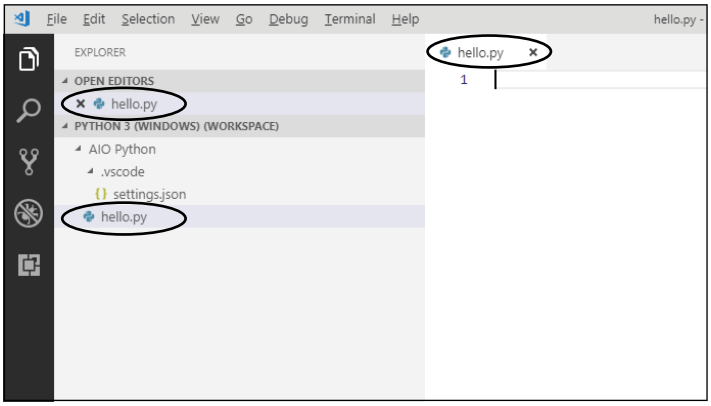


FIGURE 2-12: New hello.py file open in VS Code for editing.

Writing Python code

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

1. **Click next to the 1 in the editing area.**
2. **Type (exactly)** print("Hello World")**, and as you're typing you may notice text appearing on the screen. That 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 shows, but doesn't execute. That's because you typically don't type and run one line of code at a time. You may also notice a couple of other changes, as shown in Figure 2-13:

- The Explorer icon shows a circled 1, indicating that you current have one unsaved change.
- The hello.py name in the tab and Open Editors areas show a dot, indicating that you have unsaved changes in the file.

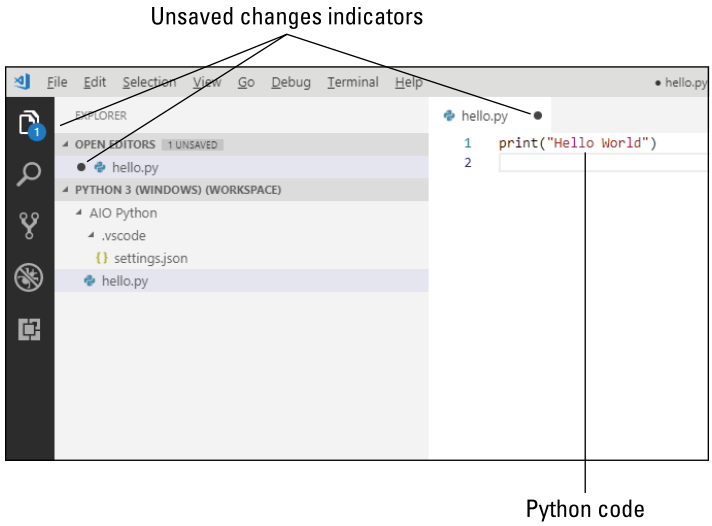


FIGURE 2-13: hello.py contains some python code and has unsaved changes.

Saving your code

Code you type in VS Code is saved automatically. There are a couple of 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 Command+S on a Mac.

Personally, we prefer to use AutoSave, which automatically saves changes you make, so that you don't have to remember. To enable Auto Save, just choose File➪Auto Save from VS Code's menu bar. When you see a checkmark next to Auto Save on the File menu, that means Auto Save is turned on, so you don't have to remember to save every change. If you decide you no longer want to use Auto Save at any time in the future, just choose File➪Auto Save again from the menu bar to remove the checkmark and turn Auto Save off.

Running Python in VS Code

To test your Python code in VS Code, you need to run it. The easiest way to do that in VS Code is probably 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.

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 in Figure 2-15. This is not the most exciting app in the world, but at least now you know how to

write, save, and execute Python programs in VS Code, and that's a skill you'll be using often as you continue through this book and through your Python programming career.

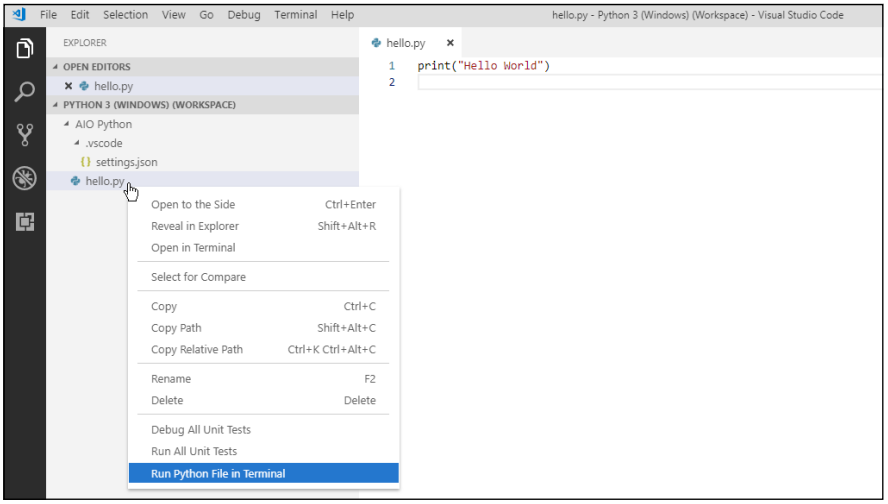


FIGURE 2-14: Run hello.py.

FIGURE 2-15: Output from hello.py.

TECHNICAL STUFF

If you are 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 really need it (for whatever reason), you might as well click Use Command Prompt if you see that option so that prompt won't keep pestering you in the future.

Simple debugging

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

Before you even attempt to run some code, you may see several indicators on the screen indicating that there is 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 show in red next to the filename in the Explorer bar.
- The total number of errors will show next to the circled X at the bottom left corner of the CS Code window.
- The bad code will likely have a wave red underline beneath it.

Figure 2-16 shows an example where we typed PRINT in all uppercase, which is not allowed in Python. Python is case sensitive, so when we show a command to type in lowercase, that means you have to type it in lowercase, too.

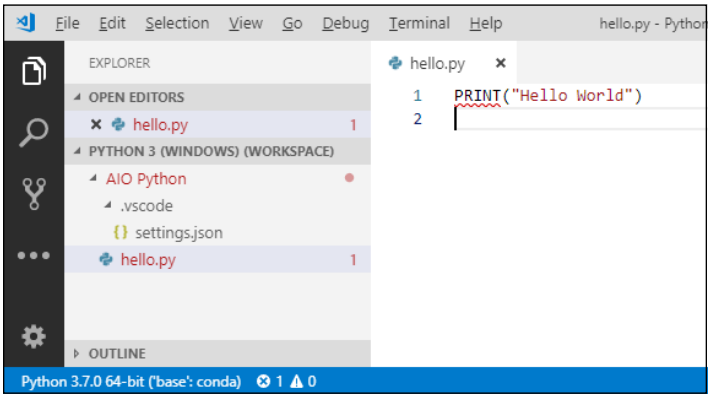


FIGURE 2-16: PRINT is typed incorrectly in hello.py.

To run the file in Terminal, you'll have to fix the error. Touching the mouse pointer to the word with the red wavy line below will give you a brief (though highly technical) description of the problem. In the example shown in the image, we just typed *PRINT* (uppercase) rather than *print* (lowercase). So, to fix the error we would just replace *PRINT* with *print* and then save the change (unless you've turned on Auto Save). Then you can right-click and choose Run Python File in Terminal to run the corrected code.

The VS Code Python debugger

VS Code also has a built-in debugger that can help when working with more complex programs and also provides another means of running Python programs in

VS Code. We won't be doing anything super complex right now. But there certainly is 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.**
2. **Next to No Configurations near the top of the pane click the Gear icon (which is probably showing a red dot right now, because you haven't specified a debugger yet). A new file named launch.json opens to the right.**
3. **Click open the drop-down menu at the top of the pane and choose Python: Current File (Integrated Terminal) (AIO Python), as shown in Figure 2-17.**
4. **Close launch.json by clicking the X on its tab.**

From now on, as an alternative to using the right-click method to run Python code, you can use the Debug pane. But first, you have to know which file will run. For now that's easy because hello.py is the only one we have. In the future, when you may have several open, you can tell which one is open because it appears in the editing area to the right of the explore pane. To run it, from the Debug pane, click the Start Debugging arrow near the top of the Debug pane, which shows as a green triangle.

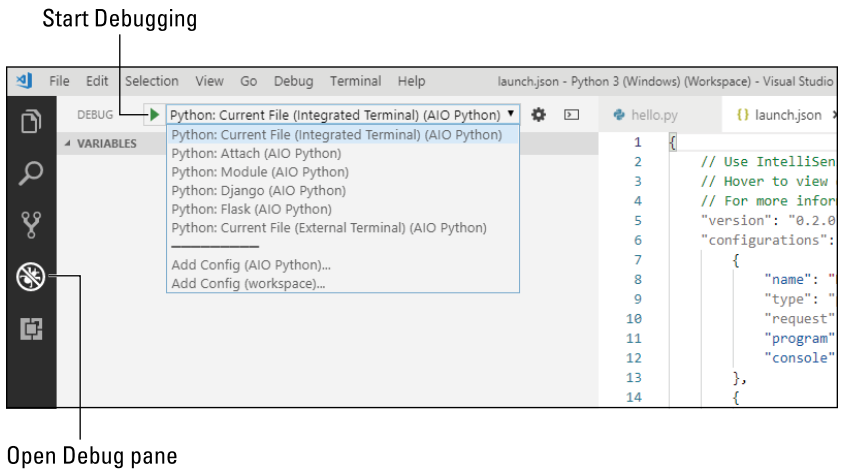


FIGURE 2-17: VS Code Debug pane.

When you click Start Debugging, the Python code will run as it did when you chose Run Python File in Terminal. However, it will take a little longer, and there will be more text output on the screen in Terminal. However, so long as there aren't any

actual errors, you should see the output under all of that (Hello World) followed by the command prompt for your operating system.

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

- 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 that file, click Debug, and then click Start Debugging at the top of the Debug pane.

If you can remember those two things, you're well on your way to learning Python, and you have a good environment in which to work.

Writing Code in a Jupyter Notebook

In Chapter 1 you learned about Jupyter notebooks as another way to write and run Python code. In this chapter, we'd like to build on what you learned there by showing you how to create, save, and open Jupyter notebooks. You can, of course, save your Jupyter notebooks wherever you want using any filenames you want. For our working example here we'll create subfolder named Jupyter Notebooks inside your AIO Python folder just to keep everything together.

Creating a folder for Jupyter Notebook

A Jupyter Notebooks folder is no different from any other, so you can create it using whatever method you normally use in your operating system. We'll put ours in the AIO Python folder we created, again just to keep all the files for this book in one place. The steps to do this are as follows:

1. **Open your AIO Python folder in Finder (Mac) or Explorer (Windows).**
2. **Right-click an empty spot in that folder and choose New➪Folder (in Windows) or New Folder on a 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 from there.**
2. **On the first page, navigate to the Jupyter Notebooks folder you created in the previous section. You should see something like *The notebook list is empty* because the folder is empty.**
3. **Click New and choose Python 3.**
4. **Near the top of the new notebook that opened click Untitled, type in the new name** 01 Notebook**, and click Rename.**

That's it, the notebook is created and saved in your AIO Python folder as shown in Figure 2-18.

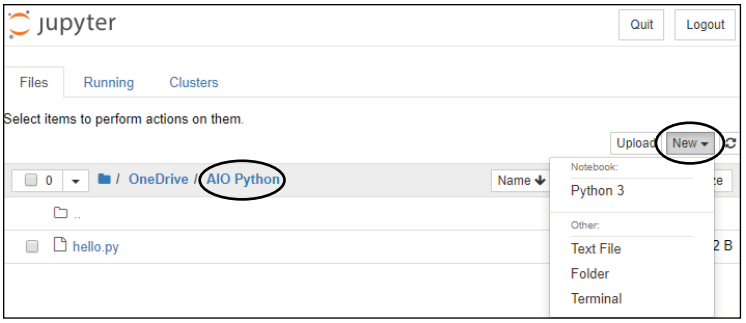


FIGURE 2-18: 01 Notebook created in Jupyter Notebook.

Typing and running code in a notebook

When your notebook is open, you can type Python code into any Code cell and text into any Markdown cell. When you see the word Code in the drop-down menu in the toolbar below the menu bar, the active cell is for typing code. To take it for a spin, follow these steps:

1. **Click in the Code cell (to the right of In [ ]: and type** print("Hello World")**and don't forget to use lowercase letters for the word *print*.**
2. **To run the code, hold down the Alt key (in Windows) or the Option key (on a Mac) and press Enter, or click the Play triangle to the left of the word *In*.**

   The output from the code appears below the cell, and another cell opens below.

Adding some Markdown text

As mentioned, you can add text (and actually pictures and video) to Jupyter notebooks. You don't need to use any special coding for typing regular text. If you want to do some formatting or add pictures and videos, you'll need to use Markdown code. Markdown is a popular markup language, something like a greatly simplified HTML. Although w can't go into a lengthy tutorial on Markdown here in a Python book, we can tell you that it's probably actually easier to type up the Markdown content in VS Code and then copy/paste it over into a Markdown cell than it is to type directly in the Markdown cell in Jupyter. Just make sure that when you're working with Markdown, you choose Markdown from the drop-down menu in the toolbar.

Figure 2-19 shows where we added some Markdown and text to a Markdown cell in Jupyter.

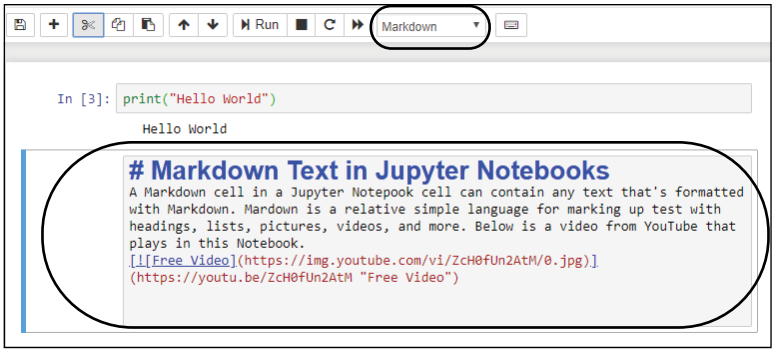


FIGURE 2-19: A Markdown cell with some Markdown code and text in it.

TIP

Alan has some free video tutorials in his online school at `https://alansimpson.thinkific.com/courses/easy-markdown-with-vs-code` if you'd be interested in learning more about Markdown.

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

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, then make your changes.

After making changes, click Run again. Note that only the cell that contains the cursor will run again. If you want to run all of the cells in a notebook, use the double triangle button a little to the right of the Run button.

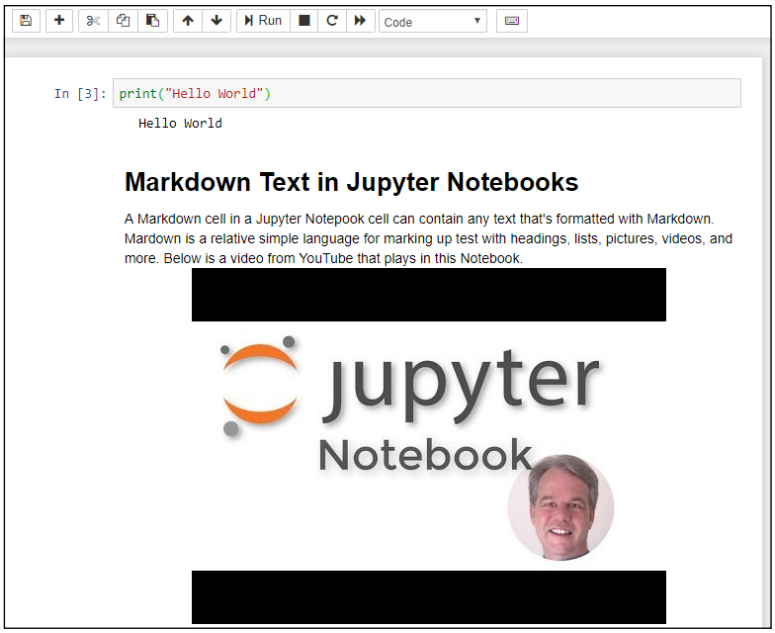


FIGURE 2-20: A Markdown cell with some Markdown code and text in it.

Saving and opening notebooks

To save a Jupyter notebook, click the Save button on the toolbar, or choose File➪Save and Checkpoint from the menu.

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

Any time that you want to reopen a notebook in the future, open Anaconda and launch Jupyter Notebook from there. Then, navigate to the file you saved and click its filename. The filename will probably show a .ipynb filename extension, as that's standard for Jupyter notebooks.

It's worth noting that when you open the AIO Python folder in VS Code, you'll see the new Jupyter Notebooks folder in the Explorer pane. Keep in mind that we put that folder there just to keep files from this book organized. There wouldn't really be any reason to open that folder in VS Code. The folder is named Jupyter Notebooks because it's just for files you create and manage in the Jupyter Notebook app.

Okay, so we've dug a little deeper in VS Code and Jupyter Notebook here, mostly so that you can save and open Python files and Jupyter Notebooks whenever you want in future chapters. All of these skills will prove useful when you start getting deeper into writing Python code. See you there!

IN THIS CHAPTER

- The Zen of Python
- Understanding object-oriented programming
- Indentations count, big time
- 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 is different in that's 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.

The Zen of Python

The Zen of Python is a list of guiding principles for the design of the Python language. (See Figure 3-1.) These principles are actually hidden in an *Easter egg* (a slang term for something in a programming language or app that's not easy to find and that serves as a bit of an inside joke to people who have learned enough Python to be able to find it). 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.**

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**, and then click Python: Select Interpreter and choose the Python 3 version that came with Anaconda.

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

The list of 19 aphorisms appears. You may have to scroll up and down or make the Terminal pane taller to see them all. But as you can see, the aphorisms are somewhat tongue-and-cheek in their philosophical rhetoric. But the general idea they express is always to try to make the code more human-readable than machine-readable.

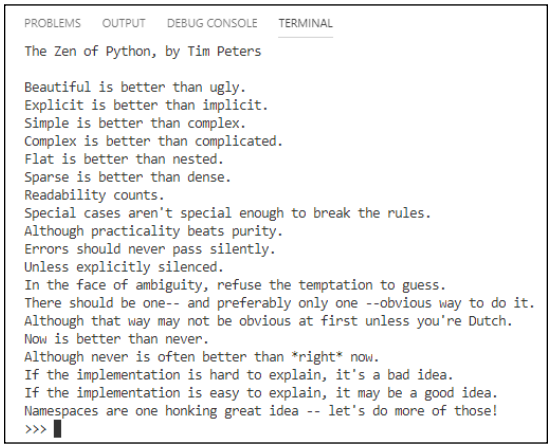


FIGURE 3-1: The Zen of Python.

The Zen 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. But it's all in fun, so don't worry about the twentieth (nonexistent) Zen . . . it won't be on any tests.

There are many other PEPs, all of which you can find on the Python.org website at `https://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 these guidelines is "readability counts" — and what they mean 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 as we go along in this book. Trying to read about it before actually 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 just by googling *pep 8*.

Truthfully, this PEP 8 business can be a kind of double-edged sword for learners. On one hand, you don't want to learn a bunch of bad habits only to discover you have to unlearn them later. On the other hand, the formatting demands of PEP 8 are so strict many learners get frustrated just trying to get the stuff to work without having to worry about blank spaces, upper-/lowercase letters, and other details.

To deal with all of this, we will be following and explaining PEP 8 conventions as we go along. You can take it a step further, if you like, by configuring PyLint to help you along. *PyLint* is a tool that comes with Anaconda that makes suggestions about your code as you're typing. Follow these steps to turn on Pylint and PEP 8 now if you'd like to take it for a spin:

1. **On the Mac menu, click Code; in Windows, click File on the menu.**
2. **Choose Preferences➪Settings.**
3. **Click Workspace Settings.**
4. **Click the three dots near the top-right of the settings and choose Open Settings.json.**
5. **Click Workspace Settings.**
6. **In the Search Settings box, type** pylint**.**
7. **Touch the mouse pointer to python.linting.enabled, and if that option isn't enabled, click it and select True to enable linting.**
8. **Scroll down and touch the mouse pointer to python.linting.pep8Enabled, click the little pencil icon that shows, and click True.**

   You'll see a couple of lines added to your Workspace Settings, each starting with *python.linting*, as shown in Figure 3-2.

FIGURE 3-2: Workspace settings with Pylint and PEP 8 enabled.

When you're done, choose File➪Save All. Then you can close the settings pages by clicking the X in their tabs. If at any time in the future you feel that the linting is just too much and is actually making it harder for you to learn, you can turn those features off by following these steps:

1. **Choose Code➪Preferences➪Settings (on a Mac) or File➪Preferences➪Settings (in Windows).**
2. **Click Workspace Settings, and then click the three dots and choose Show Modified Settings.**

   You will see Workspace settings in the non-code format shown in Figure 3-3.

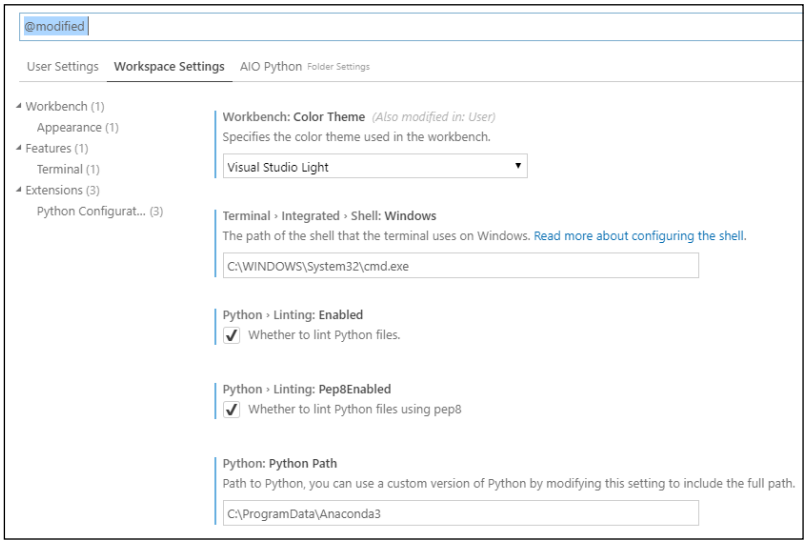


FIGURE 3-3: A different view of Workspace Settings.

3. **If you want to disable just PEP 8 (which is the one that drives most beginners a little crazy), click the checkmark for Python ➪ Linting: Pep8Enabled.**

   Try things with PEP 8 disabled. If the linting still feels like it's too much, you can repeat the steps and clear the checkmark next to Python ➪ Linting:Enabled.

Object-Oriented Programming

At the risk of getting too technical/computer science-y, we should mention that there are different kinds of languages and different approaches to designing languages. Perhaps the most successful and widely used model is what's called object-oriented programming, or OOP for short. It's a design philosophy that tries to mimic the real world in the sense that it consists of objects, with properties, and 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, size, and so forth that 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. We can drive them, steer them, speed them up, slow them down, control the inside temperature, and more using *controls* within the car that we can manipulate with our hands.

An object in an object-oriented programming language isn't a physical thing, like a car, because it exists only inside a computer. However, you can have a class (which you can think 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 controlled through the controls that all have in common, much as all cars are controlled by the steering wheel, brakes, accelerator, and gearshift.

Python is very much an object-oriented language. This may not be readily apparent when you first start learning because the core language consists of "controls" (in the form of words) that allow you to control all different kinds of objects. 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. This is sort of like, once 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.

Indentations Count, Big Time

In terms of the basic style of writing code, the one thing 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. In this book, we don't assume you're familiar with other languages, so don't worry if that means nothing to you. But if you do happen to have some familiarity with a language like JavaScript, you know that there's 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 notice that there are a lot of parentheses, curly braces, and semicolons in there:

```
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");})
```

That 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. Doing so isn't required in JavaScript, but it can be done to make the code a little easier for a human to read, as shown below:

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

Things are quite the opposite in Python because Python doesn't use curly braces or any other special characters to mark the beginnings and ends of blocks of code. The indentations themselves mark those. So those indentations aren't at all optional — they are in fact required, and they have a considerable impact on how the code runs. But the upside is, 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 notice at the top of the Python code there is a line that starts with the word `import`. Those are very 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 relatively simple, clean, core language. That's the part you really need to learn first. In addition to that core language, there are many, many modules out there 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 really 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 kind of specific application like science or numbers or artificial intelligence or working with dates and time or . . . whatever. The beauty of it is that somebody else (probably a lot of people) spent a lot of time creating, testing, and fine-tuning that module so you don't have to. You just have to import whichever modules you want into your own .py file, and use the modules' capabilities as instructed in the documentation that's available for each module.

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

```
import random
```

As it turns out, the core Python language has nothing built into it to generate a random number. Although we could probably figure out some way to make one, there's no need to because somebody else has already figured out how to do this and made the code freely available. Starting our program with `import random` tells the program we want to use the capabilities of that module to generate a random number. Then, later in the program, we generate a random number between 1 and 8 with this specific line of code:

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

Using the existing module saves me from having to reinvent the wheel trying to figure out how to make a random number from the core language. And there are literally hundreds of free modules for Python, which means you pretty much never have to reinvent any wheels. You just need to know which module to import into your program.

Now, you may be wondering where all the modules are located and where you can get them. Well, truthfully, they're all over the place online. But chances are, you will never need to go and download one because you already have all the most widely used modules in the world. You have them because they came along with Anaconda when you downloaded and installed that. To see for yourself, follow these steps:

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

Those things you see to the right are actually Python modules that are already installed on your computer and ready for you to import and use as needed. (See Figure 3-4.) As you scroll down through the names, you'll see that you already have a ton of them. In the right column, you even get to see which version of the module you have.

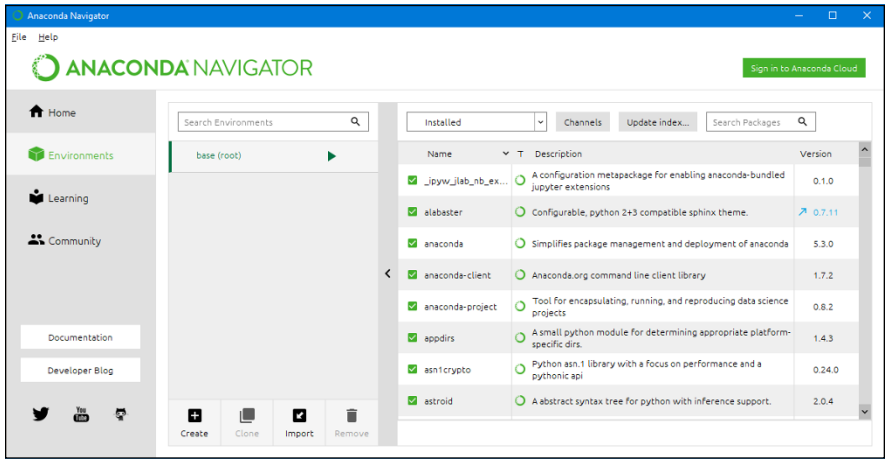


FIGURE 3-4: Installed modules.

You may also notice that some of the version numbers in the right column are colored and have an arrow showing. These represent modules that may have a more recent version available for you to download. As with programming languages, modules evolve over time and version as their authors improve things and add new capabilities. You're not required to always have the latest version, though. If the version you have is working, you can stick with that.

One of many nice things about Anaconda is that in order 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 you want to download. In fact, you can click as many as you want. Then click Apply at the bottom-right. Anaconda does all the dirty work of finding the current module, determining whether there actually is a newer version available, and then downloading that version, if it's available.

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

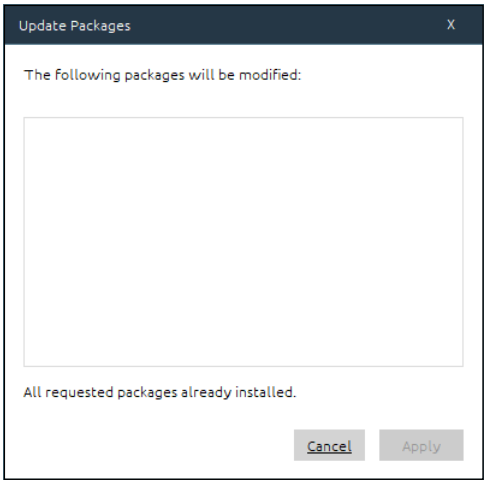


FIGURE 3-5: All our packages are installed and up-to-date.

Syntax for importing modules

As we already 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]
```

When you see such a chart, always remember four things:

- The code is case-sensitive, meaning you must type `import` and `as` using all lowercase letters, as shown. It won't work if you use uppercase letters.
- Anything in italics is a placeholder for specific information that you'll supply in your own code. For example, there are dozens of modules available to you. In your code, you must replace *`modulename`* with the exact name of the module you want to import.
- Anything in square brackets is optional, meaning you can type the command with or without that part.
- 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` any place you type Python code: the Python command prompt (`>>>`), in a .py file, or in a Jupyter notebook. In a .py file you should always put the `import` statements first, so their capabilities are available to the rest of the code.

Using an alias with modules

As mentioned in the previous section, you can assign an alias, a "nickname," to any module you import just by following the module name with a space, the word `as`, and then a name of your own choosing. This is usually a short name that's easy to type and remember, so you don't have to type the long name every time you want to access the module's capabilities.

For example, instead of typing `import random` to import that module, we could import it and give it a nickname like `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`, as in the example below:

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

It may not seem like a big deal in this short example. But you may come across some modules that have lengthy names, and your code requires referring to that module in many different places. Having the alias name available lets you type just that short name, you don't need to type the full name.

In previous chapters, we promised lots of hands-on in this book. Telling you all these facts without a lot of guided hands-on practice amounts may seem like cheating, then. We've done so, however, because these bits of background knowledge will help you make sense of things as you learn Python, and should help with seeing the big picture and remembering things. But now, without any further ado, it's time to really apply all you've learned so far and start getting your hands dirty with some real Python code. See you in the next chapter.

IN THIS CHAPTER

- Open the Python app file
- Typing and using Python comments
- Understanding Python data types
- Doing work with Python operators
- Creating and using variables
- What syntax is and why it matters
- Putting code together

Chapter 4
Building Your First Python Application

So you want to build an application in Python, huh? Whether you want to code a website, analyze data, or create a script for automating something, this chapter gives you all 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 just applications or just programs or even apps for short. To create apps, you need to know how to write code inside a code editor. You also need to start learning the language (Python, in this book) in which you'll be creating those apps.

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 create your code. You then learn all about the various data types, operators, and variables that 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 even how to comment your code so that you and others can understand how you built it and why.

Are you ready?

Open the Python App File

As we mentioned in the preceding chapters, we'll be using the ever-popular Visual Studio Code (VS Code) editor in this book to learn Python and to create Python apps. Here we assume you've already set up your learning/development environment as described in this chapter, and know how to open the main tools, Anaconda Navigator and VS Code. To follow along hands-on 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 the previous chapter.**
4. **Select all the text on the first line and delete it, so we can start from scratch here.**

At this point, you should have `hello.py` open in the editor as shown in Figure 4-1. If you still have any other tabs open from before, close those now by clicking the X in each.

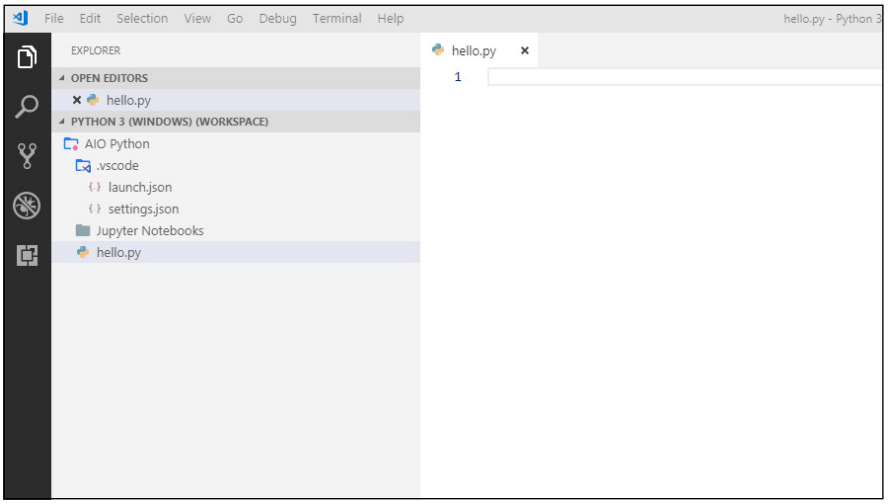


FIGURE 4-1: The `hello.py` file, open for editing in VS Code.

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, then 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.

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, the text must

- Start with a pound sign
- Or be enclosed in triple quotation marks

If it's a short comment (if it doesn't extend to two or more lines), the leading pound sign is sufficient. Often you'll see that pound sign followed by a space, as in the example below, but that space is optional:

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

To type a Python comment into your own code

1. **Click next to the 1 in VS Code under the hello.py tab and type** # This is a Python comment in my first Python app**.**
2. **Press Enter.**

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

Although you won't use multiline comments just yet, be aware that you can type longer multiline 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.
```

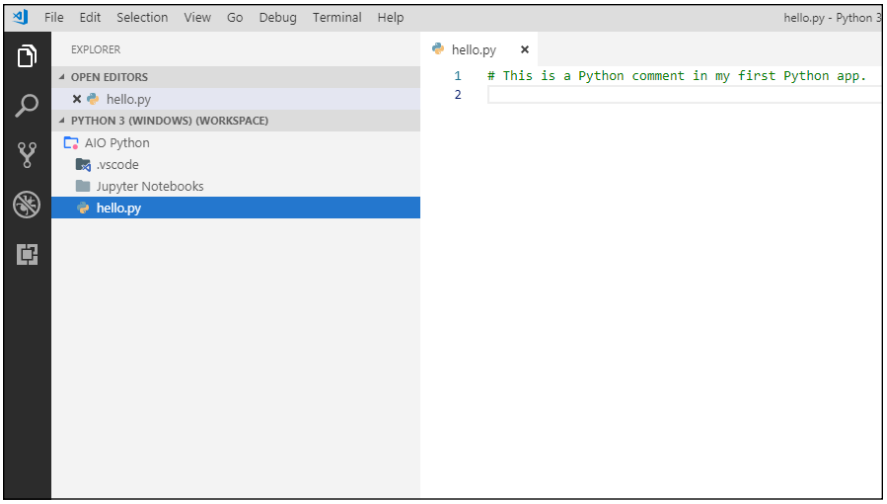


FIGURE 4-2: A comment in `hello.py`.

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

In case you're wondering, there is no limit to how many comments you can put 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. Right now, all we have is a comment, so even if we did run this code, nothing would happen because code is for human readers, not computers. Before you go hands-on and start typing actual code, we 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 (that is, letters and words). But there's a big difference between numbers and text in a computer because with numbers,

you can do arithmetic (add, subtract, multiple, divide). For example, everybody 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 — a *scalar value* in programming jargon. You can't really buy a *snorkel apples* because a snorkel is a thing, it's not a quantity, number, or 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 contain no 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	Okay?	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 one.
$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 actually 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 like 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* for short, is just 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 happen to 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 3 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 float, so being familiar with those is sufficient for just about everyone. Though in the interest of being accurate we should point out 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 absolutely no idea what we're talking about, then you're a normal person because only people who are really "out there" in math land care about these. 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 for computers in general). It's called a *string* because it's a string of characters (letters, spaces, punctuation marks,

maybe some numbers). To us, a string usually has some meaning, like 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 if it's not something on which you 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 quotation marks (“) or singles (‘). All the following are valid strings:

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

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

A word of caution. If a string contains an apostrophe (single quote), then the whole 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 second 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, then the whole thing should be enclosed in single quotation marks to avoid confusion. For example:

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

So 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 that string.

So what if we 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 can happen, so they came up with an escape. In fact, the things you use are called *escape characters* because, in a sense, they allow you to escape (avoid) the "special meaning" of a character like 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.

Using that last example, we could enclose the whole 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, we could enclose the whole 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 add a line break to end a line, on the screen where a user is viewing it (the *user* being anybody who uses some 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!
```

. . . because each \n would be converted to a line break, ending the line there:

True/false Booleans

There is a third data type in Python that 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 a little odd to have a data type for something that can only be True or False. However, because of the way computers do their work, it's actually efficient to make True/False its own data

type. A single bit, which is the smallest unit of storage in a computer, is all that's required to store a value that can only be True or False.

In Python code, people store True or 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're not enclosed in quotation marks (as a string would be), and they're not numbers. Also, the *initial cap* is required. In other words, the first letter has to be capitalized and all the remaining letters after that have to be lowercase.

Doing Work with Python Operators

You use computers in a couple of ways. One way is to simply *store* information, like a file cabinet but without any paper. As we discussed in the previous section, with Python and for 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 to 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 just 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

As the name implies, arithmetic operators are for doing arithmetic; addition, subtraction, multiplication, division, and others. 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" in the sense that 3**2 is 3^2 or 3 squared, which is 3*3 or 9. And of course 3**4 would be 3*3*3*3 or 81.
- *Floor division*, indicated by //, is integer division in that anything after the decimal point is truncated (ignored). The term *truncated* in this sense means "cut off," without any rounding. For example, in regular division 9/5 is 1.8. But 9//5 is just 1 because the .8 is just chopped off, it doesn't even round up the 1 to a 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. They are decisions based on absolute facts based on comparisons. The operators Python offers to help you write code that makes decisions are listed in Table 4-3.

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. For now, we just want to show them so that in case you're ever looking at other people's code, you'll have some sense of what they mean.

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

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 full

Python style guide 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, then type the operator, 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 exact meaning of the code. Instead, notice 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'll 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 browse Amazon today, you can see your name and "member since" date appear on their home page, as shown in Figure 4-3.

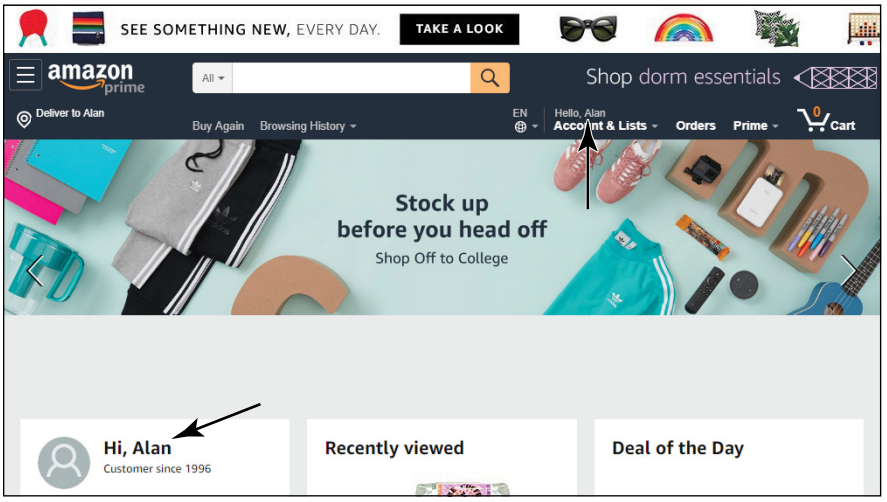


FIGURE 4-3: The user's name and "Member since" date on Amazon's home page.

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. So most of what's on that page is probably *literal* — meaning everybody 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 (that is, only the information that is variable). A variable, in Python, is simply a storage place for information that can change at any time. But in your code it's represented by a variable *name* rather than a place.

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 3 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.

So, if you consider Quantity and Unit Price to be *variables* in which you can store numbers, you get the correct extended price when you multiple those two numbers no matter what numbers you use to replace Quantity and Unit Price.

Creating valid variable names

In our explanation of what a variable *is* we used names like Quantity and Unit Price, and this is certainly fine for a general example. Those are names we just made up. In Python, you can also make up your own variable names. But the rules are more stringent than when making them up in plain English. Python variable names have to conform to certain rules to be recognized as variable names. The rules are

- The variable name must start with a letter or underscore (_).
- After the first letter you can use letters, numbers, or underscores.
- Variable names are case-sensitive, so once you make up a name, any other reference to that variable must use the same uppercase and lowercase letters as the name you originally made up.
- Variable names cannot be enclosed in, or contain, single or double quotation marks.

» PEP 8 style conventions recommend using only lowercase letters in variable names, use an underscore to separate multiple words in the variable name.

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 caps* formatting — whereby the first letter is made lowercase and new words are capitalized instead of separated by spaces — is pretty common, even in Python. For example, `extendedPrice` or `unitPrice` are Camel caps terms that are both technically valid and will not make your program fail. But experienced Python purists sometimes get that "disgusted" look on their face when they see names like these in you 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 syntax (order of things) as follows:

```
variablename = value
```

As mentioned, the *variablename* is the name you make up. You can use *x* or *y*, like people often do in math, but in larger programs, it's a good idea to give your variables more meaningful names, like `quantity` or `unit_price` or `sales_tax` or `user_name`, so that you can always 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 in the *assignment operator* is so named because it assigns the value (on the right) to the variable (on the left). For example, here . . .

```
x = 10
```

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

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. Let's go hands-on and try some simple examples just to get the hang of it. If you still have VS Code open with that one comment showing, 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 #** This variable contains an integer**.**
2. **Press Enter, type** quantity = 10 **(don't forget to put a space before and after the = sign), and press Enter.**
3. **Type #** This variable contains a float **and press Enter again.**
4. **Type** unit_price = 1.99 **and press Enter again (don't type a dollar sign!).**
5. **Type #** This variable contains the result of multiplying quantity times unit price **and press Enter.**
6. **Type** extended_price = quantity * unit_price **(again with spaces around the operators) and press Enter.**
7. **Type #** Show the results **and press Enter.**
8. **Type** print(extended_price) **and press Enter.**

Your Python app creates some variables, stores some value in them, and even calculates a new value, `extended_price`, based on the contents of the quantity and `unit_price` variables. The very last line displays the contents of the extended-price variable on the screen. Remember, the comments don't actually *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 red wavy lines near where there is an error, or possibly green wavy lines if it's just a stylistic error such as an extra space or an omitted Enter at the end of a line. If yours does have errors, make sure that you understand that when typing code, you *have to* be accurate. You can't type something that looks sorta 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 any concept of "context," and so they will generally just not work properly if there are errors in your code.

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

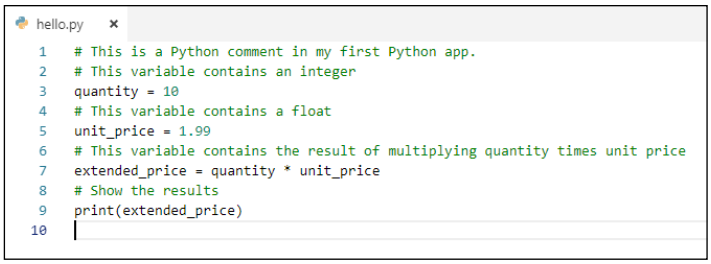


FIGURE 4-4: Your first Python app typed into VS Code.

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 previous chapters, choose File, then choose Save.

Running your Python app in VS Code

Now we can run the app and see if it works. And an easy way to do that is to right-click the `hello.py` filename in the Explorer bar and choose Running a Python app in VS Code is easy. Just right-click the file's name (`hello.py`, in this example) and choose Run Python File in Terminal as shown in Figure 4-5.

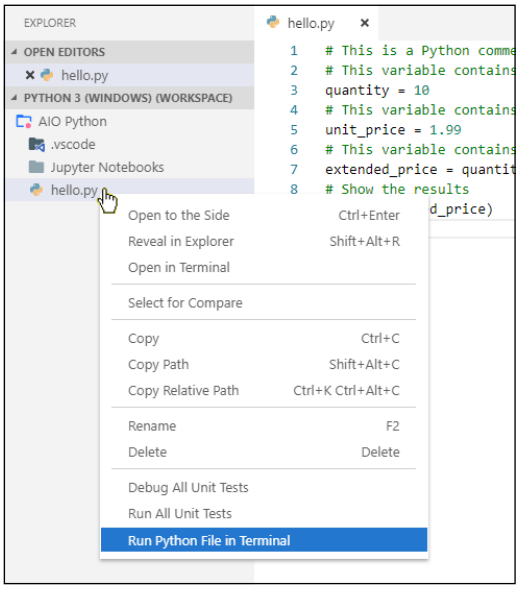


FIGURE 4-5: Right-click a .py file and choose Run Python File in Terminal.

If your code is typed correctly, you should see the result, 19.9, in the Terminal, as shown in Figure 4-6.

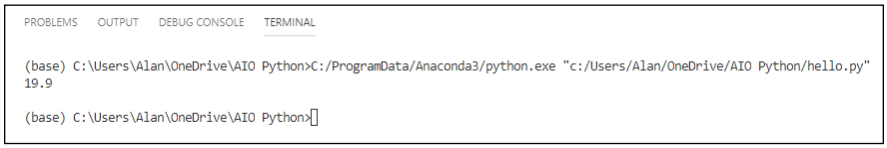


FIGURE 4-6: The 19.9 is the output from `print(extended price)` in the code.

The result of running the app appears down in the Terminal window. It certainly doesn't look like a typical phone, Mac, or Windows app. That's because you're just getting started and so need to keep things simple. The only indication that the app ran at all is the number 19.9 in the results. This is the output from `print(extendedprice)` in the code, and it's 19.9 because the quantity (10) times the unit price (1.99) is 19.9.

Suppose your app must calculate the cost of 14 items going for $26.99 each. 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 dollars 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 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 already turned on AutoSave as mentioned earlier in this chapter). Then run that app with the changes 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 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 let's just be happy with getting our apps to run.

What Syntax Is and Why It Matters

If you look up the word *syntax* in the dictionary, you'll probably see it defined as something like "the arrangement of words and phrases to create well-formed sentences in a language." In programming languages like Python, there really is no such thing as a well-formed sentence. With code, we generally refer to each line of code a s*tatement* or a *line* (of code). But the order of words is important, and there are "words" in the sense that you need a space between each word, just as you do when typing regular text like this.

Syntax is important in human languages because the 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 exact words. But the meanings are very different. The first two make it clear who kissed whom, and the last one is a little hard to interpret. Even though they all contain exactly the same words.

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 whole concept of "context" doesn't even exist for computers. So syntax matters even more when writing code to tell a computer what to do than it does for human languages.

Looking back at our earliest code, notice that all the lines of actual code (not the comments, which start with #) follow the syntax . . .

```
variablename = value
```

. . . where *variablename* is some name we made up, and *value* is something we are storing in that variable. It works because it's the proper syntax. If you try to do it like this . . .

```
value = variablename
```

. . . it won't work. For example, this is the correct way to store the value 10 in a variable named `x`:

```
x = 10
```

It may seem you could also do it this way . . .

```
10 = x
```

. . . but that won't work in Python. 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 that looks something like this:

```
  File ".../AIO Python/hello.py", line 10
    10=x
    ^
SyntaxError: can't 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
```

The Python language consists of lines of code, each one ending with a line break or semicolon. In other words, this is three lines of Python code:

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

It would also be perfectly acceptable to us a semicolon wherever instead of a line break, as below.

```
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)
```

Note that there is no advantage or disadvantage to doing so in terms of how the code runs. The code runs exactly the same either way. The semicolons are just an alternative to line breaks that you can use if, for whatever reason, you want to put a lot of code alone one physical line in your editor.

Note how our variables' names are all lowercase, and the words are separated by an underscore:

```
first_name
last_name
```

This is a *naming convention* used in Python — to use all lowercase letters for variable names with words separated by underscores. But note that a *convention* is not the same as a *syntax rule*. You could name the variables as

```
FirstName
LastName
```

This is perfectly okay in Python and doesn't break any syntax rules. The naming *convention* is just there to try to get programmers to follow some basic stylistic rules that make the code more readable to other programs, which is especially important when working in programming teams or groups.

So far we're looked at lines of code. There are also blocks of code or code *blocks* where two or more lines of code work together in some manner. Here is an example:

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


TECHNICAL STUFF

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 zero. If `x` *does* contain zero, then the indented line (`print("x is zero")`) executes and that's what you see on the screen. However, if `x` does not contain zero, then that indented line is skipped over and the `else:` statement executes. The indented line under else: `print("x is ",x)` executes, but *only* if the `x` doesn't contain zero. The last line, `print("All done!")`, executes no matter what, since it's not indented.

So, in other words, indentations matter a lot in Python, since only one of the indented lines above will execute depending on the value in `x`. We'll get into the specifics of using indentations in your code as we 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.

Next time you run the app, it will run and you won't get a syntax error for that line.

If you have linting and PEP 8 enabled in your Workspace settings, as described in Chapter 3, then you may see red wavy underlines or green wavy underlines on code that appears to be perfectly okay. Touching the mouse pointer to such an underline will usually show a message at the mouse pointer indicating what the problem is, as in Figure 4-7.

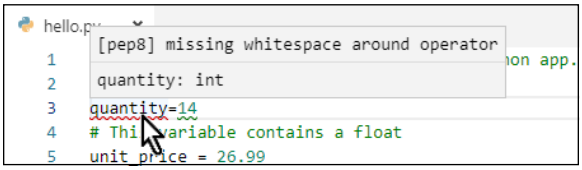


FIGURE 4-7: Touched the mouse pointer to a red wavy underline.

The first part of the message reads:

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

That's telling you that this particular error is related to Pep 8 syntax, which says you should put whitespace around operators.

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

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

Now let's suppose you fix that spacing issue but still see a green wavy underline under the 14. What's up with that? Well, to find out, simply click the green wavy underline and leave the mouse pointer sitting right there until you see an explanation, as in Figure 4-8.

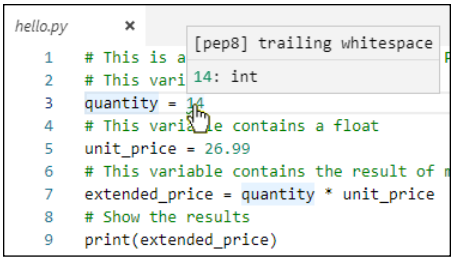


FIGURE 4-8: Touching the mouse pointer to a green wavy underline.

This time, the top message shows *[pep8] trailing whitespace* on top, and *14: int* on the bottom. Again, the bottom part is just informational, telling you that the 14 is stored as an integer. The error is the trailing whitespace. What this means is that, for whatever reason, there happens to be a blank space after the 14 on that line. You can't see it, because it's just a space. But if you click the end of that line and press Backspace until the cursor is right up to the 4 in 14, then you will have eliminated any trailing spaces, and that should clear that error.

Keep in mind that not all red errors are the same and not all green errors are the same. So don't make any assumptions. In general, red errors are likely to prevent the app from running correctly, while green errors are just stylistic errors. But you won't know, specifically, what the error is unless you click 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 that error.


TIP

If the PeP 8 errors really drive you nuts when trying to learn, remember you can go into the VS Code Workspace Settings and turn off the Python➪Linting: Pep8Enabled setting so it doesn't check your code so aggressively.

Putting Code Together

The exercises you've just completed explain how to type, save, run, change, and save again, run an app again after making your changes. All those different things define what you'll be doing with any kind of software development in any language. So you want to practice them often until they become second nature to you. But don't worry, that doesn't mean you have to do this one chapter over and over again to get the hang of it. You'll be using all these same skills throughout this book as you work your way from absolute 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
- Uncovering dates and times

Chapter 1

Working with Numbers, Text, and Dates

Computers 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 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 (called *integers*) and numbers that contain a decimal point (called *floats*). Words (textual information like 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`, but nothing else. In real life, we also have to deal with dates and times, which are yet another type of information. Python doesn't actually have any 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 it a number, and it gives 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 an = sign and the function name, followed by a pair of parentheses. Inside the parentheses you may pass one or more values (called *parameters* or *arguments*) 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 exactly 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, the Python prompt, or a .py file in VS Code), I created a variable named `x` and assigned it the value `-4`. Then I create a variable named `y` and assigned it the absolute value of `x` using the `abs()` function. Printing `x` then shows its value, `-4`, which hasn't changed. Printing `y` shows `4`, the absolute value of `x` as returned by the `abs()` function.

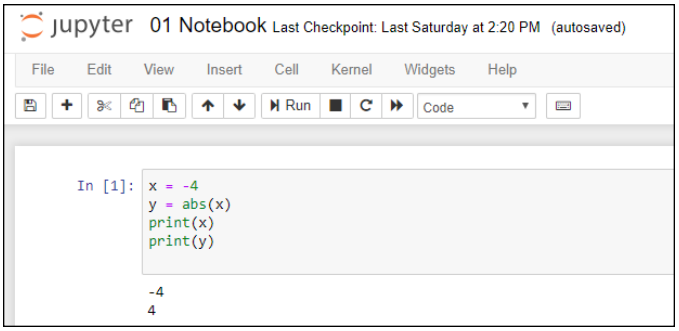


FIGURE 1-1: Trying out the `abs()` function.

Even though a function always returns one value, there are some that accept two or more values. For example, the `round()` function takes one number as its first argument. The second value 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 numbers after the decimal point. Then we created a variable named `y` to return that same number rounded to two decimal places. Then we printed both results.

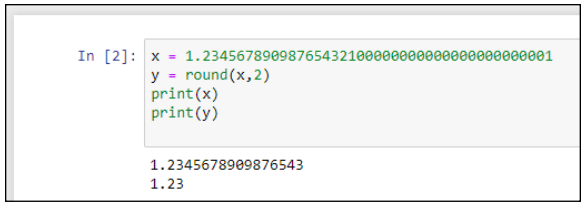


FIGURE 1-2: Trying out the `round()` function.

Python has many built-in functions for working with numbers, as shown in Table 1-1. Some of them 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 that's relevant to the kind of work you do. But if you're curious, you can always google the word *python* followed by the function name for more information. For a more extensive list, google *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 a the float data type
`format(x,y)`	Returns *x* formatted as directed by format string *y*. In modern Python you're more likely to use f-strings, as described later in this chapter
`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 point and any digits after it.
`max(x,y,z ...)`	Takes any number of numeric arguments and returns whichever one is the largest.
`min(x,y,z ...)`	Takes any number of numeric arguments and returns whichever one 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 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 of 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 converted to positive, and then the decimal point and everything to its right chopped off).

Still More Math Functions

The built-in math functions are handy, but there are 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. At the command prompt, you can just enter the command **`import math`** before using those functions at the command prompt.

One of the functions of the `math` module is the `sqrt()` function that gets the square root of a number. Because it's part of the `math` module, you cannot use it without importing the module first. For example, if you enter this:

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

. . . you'll get an error because `sqrt()` isn't a built-in function. Even if you do two commands like this:

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

. . . you still get an error because you're treating `sqrt()` as a built-in function.

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.

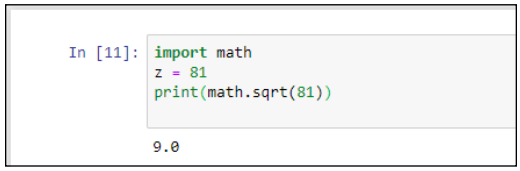


FIGURE 1-4: Using the `sqrt()` function from the `math` module.

Entering that command shows 9.0 as the result, which is indeed the square root of 81.

The `math` module offers a lot trigonometric and hyperbolic functions, powers and logarithms, angular conversions, constants like `pi` and `e`. We won't delve into all of them since advanced math isn't all that relevant to most people. You can check them all out at any time by googling *python 3 math module functions*. Table 1-2 offers a handful of examples, some of which 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 arc cosine of *x* in radians.
`math.atan(x)`	Returns the arc tangent of *x*, in radians.
`math.atan2(y, x)`	Returns atan(y / x), in radians.
`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 . . .).

(continued)

TABLE 1-2 *(continued)*

Built-In Function	Purpose
`math.exp(x)`	Returns *e* raised to the power *x*, where *e* is the base of natural logarithms.
`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 natural 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 arc sine of *x*, in radians.
`math.sqrt(x)`	Takes any number of numeric arguments and returns whichever one is the smallest.
`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 any parentheses. As with any function, you can use those functions in expressions (calculations) or assign their values to variables. Figure 1-5 shows some examples of using functions from the `math` module.

```
In [22]: import math
         pi = math.pi
         e = math.e
         tau = math.tau
         x = 81
         y = 7
         z = -23234.5454
         print(pi)
         print(e)
         print(tau)
         print(math.sqrt(x))
         print(math.factorial(y))
         print(math.floor(z))
         print(math.degrees(y))
         print(math.radians(45))

         3.141592653589793
         2.718281828459045
         6.283185307179586
         9.0
         5040
         -23235
         401.07045659157626
         0.7853981633974483
```

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 getting numbers to display in formats that are familiar to us humans. For example, we may prefer it display $1,234.56 rather than 1234.560065950695405695405959. As of version 3.6 of Python, f-strings seem to the fastest, easiest, and most preferred method of achieving this.

Formatting with f-strings

F-string formatting is relatively simple to do. All you need is a lowercase or uppercase *f* followed immediately by some text or expressions enclosed in quotation marks. Something along these lines:

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

The `f` before the first quotation mark tells Python that what follows is a format string (f-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. The expression part is a placeholder for what's actually going to show there 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 a variable named `username`, as in Figure 1-6.

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

Here is another example where we have 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, is:

```
Subtotal: $1499.7
```

That `$1499.7` isn't really 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 example below:

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

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

Showing dollar amounts

To get a comma to show in the dollar amount and the pennies to be 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 being shown.

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

```
:,
```

Using our current example, this would be:

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

Executing this statement produced 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). So here's how the code looks to show the number with commas and two decimal places:

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

Here is how this code looks when executed:

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

Perfect! That's exactly the format we want. So any time 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. This 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, then 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 show that number to a human, simply displaying it with a `print()` function shows 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, don't use the `f` for "fixed numbers." Instead replace that with a `%` sign, like this:

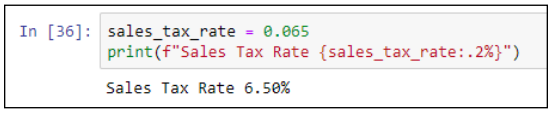
```
print(f"Sales Tax Rate {sales_tax_rate:.2%}")
```

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

In both of the preceding 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%}")
```

FIGURE 1-7: Formatting a percentage number with .2%.



. . . shows this output when executed:

```
Sales Tax Rate 6.5%
```

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

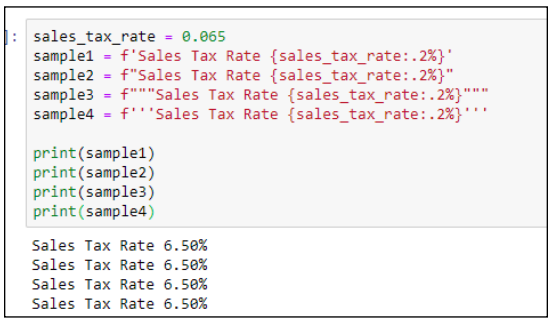
```
print(f"Sales Tax Rate {sales_tax_rate:.9%}")
```

. . . displays the percentage with nine digits after the decimal point.

```
Sales Tax Rate 6.500000000%
```

Your format string doesn't have to be one short thing inside the parentheses of a `print()` function. You can store it to a variable too, then print the variable. The format string itself is like any other string in that it must be enclosed in single, double, or triple quotation marks. It doesn't matter which you use as the outermost quotation marks on the format string, the output is the same regardless, as you can see in the example shown in Figure 1-8.

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



TIP

For single and double quotation marks, use the ones on the keyboard key that shows both kinds of quotation marks. 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.

Making multiline format strings

If you want to have line breaks in your format strings for multiline output, you have a couple of choices:

- **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 any 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:** If you use triple quotation marks around your format string, then 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 all the same places.

  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 all the numbers so all 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.

```
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.

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

```
:>20
```

. . . says "make this output 20 characters wide, with the content right-aligned."

In the last example in the previous section, we were able to get all the dollar amounts left-aligned in the output by making then left-aligned in the format string. But because the numbers aren't all the same width, they're not right-aligned. To get around that, in Figure 1-10 we used > to make each of the dollar amounts right-align a width of 9. The printed output shows the numbers are correctly right-aligned.

```
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? This is because 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.

Getting around this issue is a little more complicated than you may imagine, because you can only use the `,.2f` formatting 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 won't be a number anymore so the `,.2f` won't work. Complicated, however, doesn't mean impossible; it just means inconvenient. We 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, here is how we could do the subtotal variable:

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

The `subtotal` is a number calculated by multiplying the `quantity` times the `unit_price`. The `s"{subtotal:,2f}"` formats that number in the fixed two-decimal-places format with commas in the thousands places, like this:

```
1,598.40
```

This is a string rather than a number, because an f-string always produces a string. The app sticks a dollar sign on the front of that string using `"$"+`. So you end up with a string that looks like `$1,598.40`. We put this 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.) Later in the format string we just give this a width and right-align it to that width using `>9`, like this:

```
Subtotal:  {s_subtotal:>9}
```

REMEMBER

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

Figure 1-11 shows the whole kit-and-caboodle, 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 with imaginary numbers, Python has the stuff you need to get the job done. But keep in mind that you don't need to learn these things to use Python or any other language. You would only use these if your actual work (or perhaps homework) requires it. We'll start with binary, octal, and hex numbers, which are used frequently in computer science.

Binary, octal, and hexadecimal numbers

If your work requires dealing with base 2, base 8, or base 16 numbers, you're in luck with Python because it 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()`

In all honesty, 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 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 go to your favorite search engine and search for *binary number* or *octal*, or *decimal*, or *hexadecimal*.

If you do need to work in other numbering systems for computer science homework or whatever, you can use the various functions to convert between numbering systems. You can do so right 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, this gets you the hexadecimal equivalent of the number 255:

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

The result is `0xff`, where the `0x` is there just to indicate 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.

TECHNICAL STUFF

In case you're wondering about the `print("\n")` in Figure 1-12, it displays a line break, which produces the blank line you see in the output.

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 number x whose square equals -1, so that's why it's called an *imaginary number*.

Some branches of math actually use the lowercase letter i to indicate an imaginary number. But Python uses j, as does electrical engineering, where 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 syntax:

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

Replace *`real`* with the real part of the complex number, 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`. Using a `print()` function to display the contents of `z`, like this:

```
print(z)
```

. . . displays the imaginary number `(2-3j)`.

You can tack `.real` or `.imag` into an imaginary number to get the real or imaginary part. For example,

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

. . . produces `2.0`, which is the real part of the number `z`, and

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

. . . returns `-3.0`, which is the imaginary part of `z`.

Once again, if none this makes any sense to you, don't worry about it. It's not required for learning or doing Python. Python simply offers complex numbers and those functions for people who happen to require the use of such numbers.


TECHNICAL STUFF

If your work requires working with complex numbers, google *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," which has no numeric meaning or value. (We discuss the basics of strings in Book 1, Chapter 4.)

Concatenating strings

You can join strings together using a + sign. This is commonly called *string concatenation* in nerd-o-rama world. One thing that often catches beginners off-guard is the fact that the computer doesn't know a "word" from a bologna sandwich. So when you join strings together, it doesn't automatically put spaces where you'd expect them. For example, in the following code, the `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)
```

Running this code to print the contents of the `full_name` variable reveals that Python did indeed join them together in one long string:

```
AlanCSimpson
```

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

Because Python won't automatically put in spaces where you think they should go, you'll 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 this:

```
""
```

. . . you won't get a space in your string either. You can, of course, 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 we put a space between `first_name` and `last_name`. We also stuck a period and space after `middle_init`. When we display the contents of that `full_name` variable, it looks more like the kind of name we're used to seeing.

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

The output of that code is:

```
Alan C. Simpson
```

Getting the length of a string

To determine how many characters are in a string, 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 we 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))
```

Here is the output from that code, when executed, which makes perfect sense when you understand how `len()` measures the length of strings as the number of characters (including spaces) contained within 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 zero, not one. This makes no sense to us humans. But the reasons for doing it that way have to do with what's most efficient for the way computers work. 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.

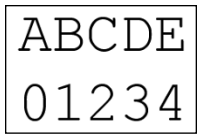


FIGURE 1-13: Character positions within a string start at zero, not one.

Table 1-4 summarizes the Python 3 operators for working with strings. Figure 1-14 shows examples of trying things out, just playing around with them in Jupyter Notebook.

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 within 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) item of string `s`.
`max(s)`	The largest (highest) item 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` let you limit the search to the characters from `i` to `j`.
`s.count(x)`	The total number of times string `x` appears in larger string `s`.

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

- The first character is always number 0.
- Every space counts as one character, so don't skip over 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 also notice 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 ASICC 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 they are essentially invisible, like when you hold down the Ctrl key and press another key.

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))` returns `A` because *A* is character 65 in the ASCII chart.

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.

Manipulating strings with methods

Every string in Python 3 is considered a *str object*. Yes, that's pronounced like *string object*; the *str* is there to distinguish it as the current, new way of doing things, as opposed to the older method called the *string object*. It's just another of those crazy things that seem deliberately confusing. Just remember that in Python 3, `str` is all about strings of characters.

The str methods (also called *string methods*) are different from functions in that the syntax is:

```
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 (if required) that you need to pass to the method. 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, 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 only 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 `find` but searches backwards from the start of the string. If `y` and `z` are provided, searches backwards from position `z` to position `y`. Returns –1 if string `x` not found.
`s.rindex()`	Same as `.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 certainly play around with these methods in a Jupyter Notebook, the Python prompt, or 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.

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 google *python 3 string methods* to find out what's available.

```
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.

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 some user did something or when some 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 need to import it before you can use it. You can 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` will 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 actually an abstract base class, which is kind of 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 (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 that includes date, time, and optionally time zone information.

We preceded each type with the full word `datetime` in the preceding examples, but if you use an alias, like `dt`, then 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

`Datetime.date` is ideal for working with dates when time isn't an issue. There are two ways to create a date object: You can use the `today()` method, which gets today's date from the computer's internal clock 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:

```
# 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)
```

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

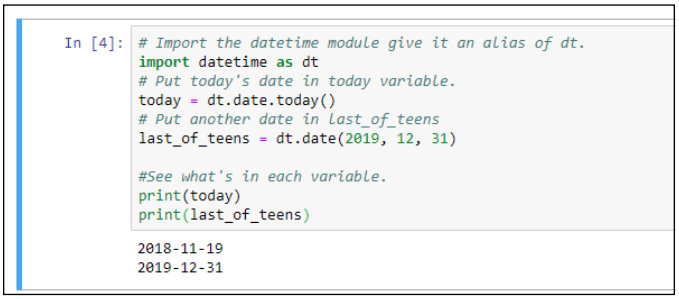


FIGURE 1-17: Experiments with `datetime.date` objects in a Jupyter notebook.

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

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

. . . produces each of the three components of that date on a separate line, like this:

```
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
%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 use `strftime` rather than f-strings for formatting dates and times, and that's certainly a valid way to do it. 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, shows 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, but the format should be like this:

```
11/19/2018
```

Table 1-7 shows a few more examples you can try out 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

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 date-time information is tailored to your time zone and takes into account the daylight savings time of your location (if it uses daylight savings time). 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.

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, using no arguments like this:

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

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

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

That second line tells you that the 00:00:00 number 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 just work with dates, and sometimes you want to just work with times. Often you want to pinpoint a moment in time using both date and 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)
```

Exactly 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 parameters shown below. The month, day, and year are required. The rest are optional and set to zero 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 back 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. Sometimes 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, 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 is 146 days between those dates; the 0:00:00 is time but because we didn't specify a time of day in either date, these are all just set to zero. 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` 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 zero.

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` using `.timedelta`. If you add the `date` and `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 off 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 because, for one thing, it's the result of subtracting one moment of time from another. Also, printing the `type()` of that data type tells us it's a `timedelta` (a duration), not an o'clock time.

Here is another example using datetimes with different dates, one being the current datetime, the other being 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 now time:

```
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 `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. For example, say you're trying to determine somebody'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 the 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 prints what's in the variable. If you actually run this code yourself, you'll see something like the following output (but it won't be exactly that 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 the `timedelta` to a number of days by tacking `.days` onto the `timedelta`. You can put that in another variable called `days_old`. Printing `days_old` and its type shows you that `days_old` is an `int`, a regular old integer you can do math with:

```
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 (/). Put that in a variable named `years_old` and print that value, as follows:

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

18
```

So there you have the age, in years, as 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 over. 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 % for division rather than / to get just the remainder after the division. Figure 1-18 shows the whole 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`.

Accounting for Time Zones

As you may know, just because it's noon, or whatever, in your neighborhood doesn't mean its noon everywhere. This is because the earth is divided into time zones so that "noon" means roughly "middle of the day" no matter where you happen to be on earth.

Figure 1-19 shows a map of all the time zones. It's not easy to see in this book, but you can easily find a larger version just by googling *time zone map* if you want a closer look.

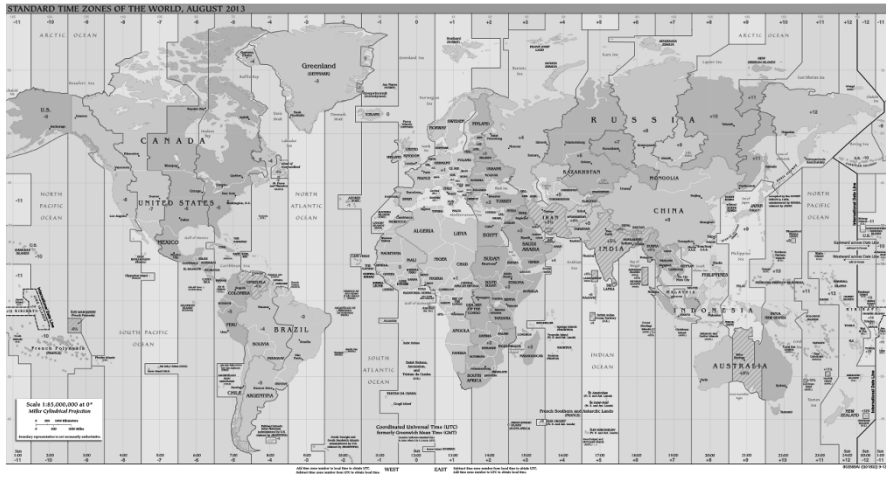


FIGURE 1-19: Time zones (larger maps available online).

Figure 1-19 shows how 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. It's 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.

These days, most people rely on the Olson Database as the primary source of information about time zones. It lists all the current time zones and locations. Feel free to Google *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 savings time.

TABLE 1-10 Sample Time Zones from the Olson Database

Time Zone	Common Name	UTC Offset	UTC DST Offset
Etc/UCT	UCT	+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
Pacific/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? It's because Python lets you work with two different types of datetimes:

- **Naïve:** Any datetime that does not include information that relates it to a specific time zone is called a *naïve datetime*.
- **Aware:** A datetime that includes time zone information is considered an *aware datetime*.

`Timedelta`s 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 that spans multiple time zones.

Working with Time Zones

When you get the time from your computer's system clock, it will be for your time zone. There just isn't any indication of what that time zone that *is*. But you can tell the difference by comparing `.now()` for your location to `.utc_now()`, which is UTC time, and then subtracting the difference, as 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: Time zones (larger maps available online).

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

Not 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 `dateutils`. 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 USA time zones.

All the USA times are standard time because nobody in the USA is on daylight savings time (DST) in November. Let's see if it's smart enough to figure out daylight savings time if we schedule an event for some time in July, when the USA is on back on daylight savings time.

In this code (see Figure 1-22), we import `datetime` and `gettx` from dateutil, as we did in the previous example. But we're not concerned about the current time. We're concerned about an event scheduled for date and time of 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, 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.

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

```
# 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.

This line of code shows the date and time, again local, since we didn't say anything about time zone. We added the word `"Local:"` to the start of the text, and put a line break at the end with `\n` to visually separate it a little 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 this 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 using this code:

```
name = event.astimezone(gettz("tzname"))
```

The first *name* is just a variable name we made up, and it could be any valid variable name. In `event.astimezone()`, the name `event` refers to the initial event time defined in a previous line. The `astimezone()` 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, the datetime of the event for five different time zones is displayed. Note that the USA time zones are daylight savings time (such as EDT). Because we happen to be on the USA east coast, and because the event in question 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 even in the general ballpark of "fun." But if you're in a pinch and really need some time zone information for your data, this shows you how to get it.

If you research Python time zones online, you'll probably find that many people recommend using the `arrow` module rather than the `dateutil` module to make everything easier. We won't get into all of that here, because `arrow` isn't part of your initial Python installation and this book is hefty enough. (If we tried to cover literally everything, you'd need a wheelbarrow to carry it around.)

IN THIS CHAPTER

- Making decisions with `if`
- Repeating a process with `for`
- Looping with `while`
- Protecting your users from errors
- Understanding contextual coding

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 there in a form that the computer can work with is certainly 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 . . . actually working with that information to create something useful or entertaining. In this chapter, we cover the most important and the 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. We use operators, such as those in Table 2-1 to make comparisons. These 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* that can allow you assess multiple comparisons before making a final decision. Those 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 google his name to learn more if you're interested.

All these operators are often used in conjunction with `if ... then ... else` type decisions to control exactly what an app or program does. To make such decisions, you use the Python *if statements.*

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* is:

```
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. Notice that neither line is indented. This means a lot in Python, as you'll see shortly. But first, let's do a couple of simple examples with this simple syntax. You can try it for yourself in a Jupyter notebook or .py file if you want to follow along.

Figure 2-1 shows a simple example where the variable named `sun` receives the string `"down."` Then an `if` statement checks to see whether the variable `sun` contains the word `down` and, if it does, prints `"Good night!"` Then it just continues on normally to print `I am here`. You can see in the output that the result is that both lines are displayed.


```
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. It's easy to forget that. If you type it wrong, it won't work as expected.

If you run the same code with some word other than `down` in the `sun` variable, then the first `print` is ignored, but the next line is still 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 variable named `sun` contains `True`, therefore the rest of that line is ignored and only the next line is executed.

That syntax, where the code to be executed when the condition proves true is on the same line as the `if` works, but 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 that are 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, the code works exactly the same as putting the code on one line. If `sun` is `down`, then `Good night!` prints before the second `print` is executed. If `sun` doesn't contain `down`, then the `print` statement for `Good night!` is skipped over and ignored.

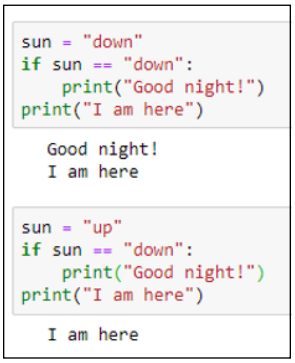


FIGURE 2-3: Result of simple `if` when the condition proves false.

If you're wondering which method is better, it depends on what you mean by *better*. If you mean *better* in terms of which method executes the fastest, then neither. You won't be able to see any speed difference when executing the code. If by *better* you mean "easier for a human programmer to read." then most people would probably lean toward 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 code under the indented lines is always executed because it's not dependent on the condition. Here is an example where we have 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 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:
```

That's 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 not `False`.

Anyway, as you can see, we start off with a `total`, a `sales_tax_rate`, and a `taxable` variable. When `taxable` is `True`, then all four lines under the `if` are executed and you end up with the output shown in Figure 2-4.

```
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.

When `taxable` is set to `False`, all the indented lines are skipped over, and the total shown is the original total without any sales tax added in, as shown in Figure 2-5.

```
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 login

So far we've looked at code examples in which some code is executed if some condition proves true. If the condition proves false, then that code is ignored. Sometimes, you may have a situation where you 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 execute. In that case, you can add an `else:` to your `if`. Any lines of code undented 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, then the program shows `Good morning!` Otherwise, it shows `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 time of day.

Now you may look at that and say "Wow, that's really 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 there are multiple `else`'s possible. That's where the `elif` statement comes into play.

Handling multiple else's 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 is possible that none of the indented code executes. 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:

```
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")
```

. . . then 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”` or `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, like this:

```
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* you put into the `light_color` variable because the else isn't looking for a specific condition. It's just playing an "if all else fails, do this" kind of role in the logic.

Ternary operations

In this book, we don't assume you're familiar with other programming languages, but we need to mention this for those readers who are familiar with other languages. Many languages have a shorthand way of doing `if...else` all on one line of code. For example, consider the following JavaScript code:

```
//JavaScript code example, won't work in Python
age=14;
beverage = (age > 20) ? "beer" : "milk";
alert("Have a " + beverage);
```

The third line is a shorthand way of saying "Put into the `beverage` variable *beer* or *milk* depending on the contents of the `age` variable" In Python you may write that as something like this:

```
age = 31
if age < 21:
    beverage = "milk"
elif age >= 21 and age < 80:
    beverage = "beer"
else:
    beverage = "prune juice"

print("Have a " + beverage)
```

The code is longer and more wordy, but easier to understand. Adding comments, which is always an option, helps a lot too, as follows:

```
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)
```

If you're wondering what the rule is for indenting comments, there is no rule. Comments are just notes to yourself, they are not executable code. So they are 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, be it games, artificial intelligence, robotics . . . whatever. But there are also cases where you need to count or perform some task over and over. A *for loop* is one way to do that. It allows 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 this 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 zero to

one 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 zero. 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
```


If you find this a bit annoying because the loop doesn't do what you meant, you can put two numbers, separated by a comma, as the range. The first number is where the counting for the loop starts. The second number is one 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 that code, the counter starts at 1, and as indicated, it stops short of the last number:

```
1
2
3
4
5
6
7
8
9
All done
```

If you really want the loop to count from 1 to 10, the range will have to be `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 printed 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 exactly the same. The only difference is we used a variable name rather than a string in the `for` loop. But it "knew" that you meant the content of `my_word`, because `my_word` isn't enclosed in quotation marks.

```
s
n
o
r
k
e
l
Done
```

Looping through a list

A list, in Python, is basically any group of items, separated by commas, inside square brackets. You can loop through such a list either directly in the `for` loop or through a variable that contains the list. Here is an example of looping through the list with no variable:

```
for x in ["The", "rain", "in", "Spain"]:
    print(x)
print("Done")
```

This kind of loop repeats once for each item in the list. The variable (`x` in the preceding example) 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.

```
for x in ["The", "rain", "in", "Spain"]:
    print(x)
print("Done")
```

```
The
rain
in
Spain
Done
```

FIGURE 2-8: Looping through a list.

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 variable `seven_dwarves` is assigned a list of seven names. Again, notice how the list is contained within square brackets. These are what make Python treat it 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, so long as the name in the first line matches the name used in the `for` loop.

```
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
do this when loop is finished
```

We put the `[do this ... ]` in square brackets because putting code above the `continue` is optional, not required. So let's say someone took an exam and we want to loop through their answers. But we have a rule that says if they leave an answer empty, then we mark it Incomplete and ignore the rest of the items in the list. Here is one where 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 the 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 there is some answer 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 code), which says `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 previous example, but instead of executing a `break` when it 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 of loops. Just make sure you get your indentations right because only the indentations determine which loop, if any, a line of code is located within. For example, in Figure 1-10 an outer loop loops through the words `First`, `Second`, and `Third`. With each pass through the loop, it prints a word, then it prints the numbers 1–3 (by looping through a range and adding 1 to each range value).

So you can see, the loops work because we see each word in the outer list 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.

```
# 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.

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 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 because of 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). This eventually increases `counter` to more than 91, which ends the loop.

The `chr()` function inside the loop just displays the ASCII character for whatever 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.

```
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.

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, it keeps printing `A`. It keeps going longer after what you see in the figure; you would have to scroll down to see how many it's done so far at any time.

If this happens to you in a Jupyter notebook, don't panic. Just hit the square Stop button to the right of Run (it shows Interrupt the kernel, which is nerd=speak for stop" when you touch the mouse pointer to it). This gets all code execution to stop in the notebook. Then you can click the curved arrow to the right of the Stop button to restart the kernel and get back to square one. Then you can fix the error in your code and try again.

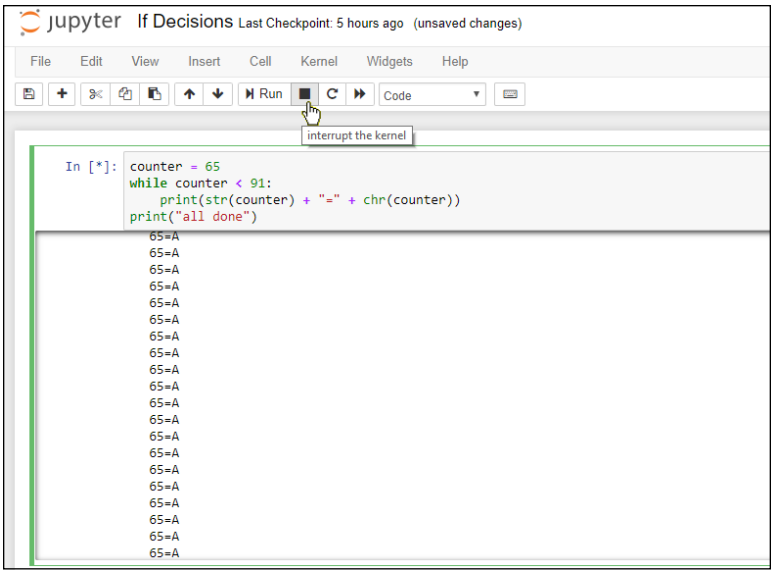


FIGURE 2-12: In an infinite loop.

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 gets a random number in the range of 1 to 999 assigned to it. Then this statement:

```
if int(number / 2) == number / 2:
```

. . . checks to see if the number is even. Remember, the `int()` function returns only the whole portion of a number. So let's say the random number that gets 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. 2 does not equal 2.5, so 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 (since 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 that's 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 as 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 even remember a situation where breaking out of a loop before its time was a good solution to the problem, so it's hard to come up with a practical example. In lieu of that, then, we'll just show you the syntax and provide a generic example. The syntax is like this:

```
while condition1:
    do this.
     ...
    if condition2
break
do this code when loop is done
```

So basically there are two things that can stop this loop. Either *condition1* proves False, or *condition2* proves True. Regardless of which of those two things happen, code execution resumes at the first line of code outside the loop, the line that reads `do this code when loop is done` in the sample syntax.

Here is an example where the program prints *up to* ten numbers that are evenly divisible by five. It may print fewer than that, though, because when it hits a random number that's evenly divisible by five, it bails out of the loop. So the only thing you can predict about it is that it will print between zero and ten numbers that are evenly divisible by five. 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 that's evenly divisible by five 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 because the random number is indeed random and not predictable (which is an important concept in many games).

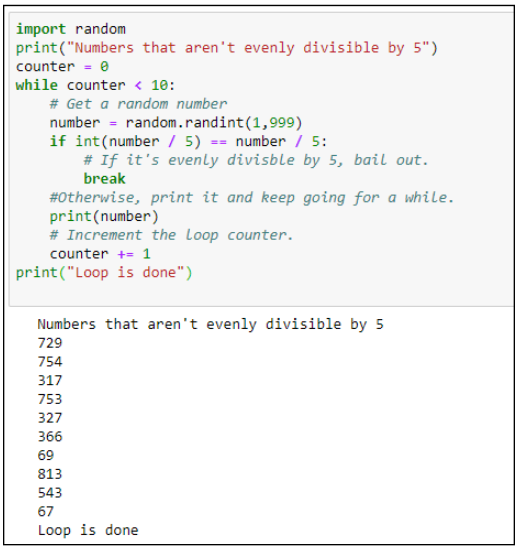


FIGURE 2-14: A `while` loop with `break`.

```
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: Same code as in Figure 2-14 on a second run.

IN THIS CHAPTER

- Defining and using lists
- Working with sets of information
- What's a tuple and who cares?
- When tuples are ideal
- How to make a tuple
- Accessing a tuple's data

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. There are also plenty of times where you work with larger sets of data, like 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 really great ways for dealing with all kinds of data collections in easy, fast, and efficient ways. You learn about those in this chapter. As always, I encourage you to follow along hands-on in a Jupyter notebook or .py file as you read this chapter. The "doing" really helps with the "understanding" part, and helps it all to sink in better.

Defining and Using Lists

The simplest data collection in Python is a list. We provided examples of these in the previous 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 = sign, just as

you would with variables. If the list contains numbers, then don't use quotation marks around them. For example, here is a list of test scores:

```
scores = [88, 92, 78, 90, 98, 84]
```

If the list contains strings then, 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 shows this on the screen.

```
['Mark', 'Amber', 'Todd', 'Anita', 'Sandy']
```

This may not be exactly what you had in mind. But don't worry, Python offers lots of great ways to access data in lists and display it however you like.

Referencing list items by position

Each item in a list has a position number, starting with zero, 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]
```

REALLY, REALLY LONG LISTS

All the lists in this chapter are short to make the examples easy and manageable. In real life, however, you may have lists containing hundreds or even thousands of items that change frequently. These kinds of lists you wouldn't type into the code directly because doing so makes the code difficult to work with. You'll more likely store such lists in an external file or external database where everything is easier to manage.

All the techniques you learn in this chapter apply to lists that are stored in external files. The only difference is that you have to write code to pull the data into the list first. This is a lot easier than trying to type every list directly into the code. But before you start tackling those really huge lists, you need to know all the techniques for working with them. So stick with this chapter before you try moving on to managing external data. You'll be glad you did.

Replace *listname* with the name of the list you're accessing and replace *x* with the position number of item you want. Remember, the first item is always number zero, not one. For example, in the first line below, I define a list named `students`, and then print item number zero from that list. The result, when executing the code, is that the name `Mark` is displayed.

```
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]
print(students[0])
Mark
```

TECHNICAL STUFF

When reading access list items, professionals use the word *sub* before the number. For example, *students[o]* would be spoken as *students sub zero*.

This 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 zero).

```
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 "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 zero with the first one on 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 there is no `scores[5]`.

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. 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 within the loop. Figure 3-2 shows a more complete example where you can see the result of running the code in a Jupyter notebook.

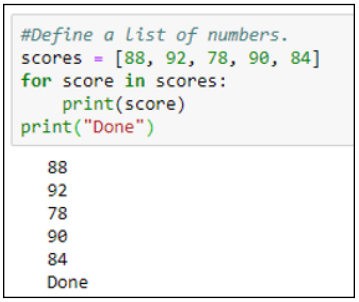


FIGURE 3-2: Looping through a list.

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 me *Bob*. Printing the contents of each variable shows 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 into a Jupyter notebook or Python prompt or whatever:

```
students = ["Mark", "Amber", "Todd", "Anita", "Sandy"]
print(len(students))
```

Running that code produces this output:

```
5
```

There are indeed five items in the list, though the last one is always one less than the number because Python starts counting at zero. So the last one, Sandy, actually refers to `students[4]` and not `students[5]`.

Adding an item to the end of a list

When you want your code to add a new 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 that reads `students.append("Goober")` adds the name Goober to the list. The line that reads `students.append(new_student)` adds whatever name is stored in the variable named `new_student` to the list. The `.append()` method always adds to the end of the list. So when you print the list you see those two new names 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 code below won't add the name Amber 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

Although the `append()` method allows you to add an item to the end of a list, the `insert()` method allows you to add 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 into 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 show 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. Just make sure you include the index number in square brackets of the 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's name 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 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 shows 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 actually execute this code and then print the list, 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 the "C" is still in the list.

```
#Create a list of strings.
letters = ["A", "B", "C", "D", "C", "E", "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, one line removes the first item (0), and another removes the last item (`pop()` with nothing in the parentheses). Printing the list shows 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 the 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 above. However, 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.

```
# Create a list of strings.
letters = ["A", "B", "C", "D", "E", "F", "G"]

# Make a copy of first list item then remove it from the list.
first_removed = letters.pop(0)
# Make a copy of last list item then remove it from the list.
last_removed = letters.pop()

# Show the new list.
print(letters)
# Show what's been removed.
print(first_removed + " and " + last_removed + " were removed from the list.")
```

```
['B', 'C', 'D', 'E', 'F']
A and G were removed from the list.
```

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 zero. 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. The letter *C* has been deleted, which is the correct item to delete because letters are numbered 0, 1, 2, 3, and so forth.

```
['A', 'B', 'D', 'E', 'F', 'G']
```

You can also use `del` to delete an entire list. Just don't use the square brackets and the index number. For example, the code in Figure 3-6 creates a list 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.

```
# 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; however, 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()`. This same 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 also added one to count the *F*'s, not using any variables. We just counted the *F*'s right in the code that displays the message. There are no *F* grades, so this returns zero, as you can see in the output.

```
# 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, remember 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, based on index number, of an item in a list. The syntax is:

```
listname.index(x)
```

As always, replace *`listname`* with name of the list you want to search. Replace *x* what whatever you're looking for (either as a literal or as a variable name, as always). Of course, there's no guarantee that the item is in the list, and even if it is, there's no guarantee that the item is in the list only once. If the item isn't in the list, then an error occurs. If the item is in the list multiple times, then the index of 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 index of a nonexistent list item.

An easy way to get around that 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 is as 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 of them 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 of this.

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, and the code is perfectly fine.

```
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, so just so it's a little easier to see what's going on and so you don't have to deal with so many commas in one line of code. 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, as shown in Figure 3-10.

```
]: # 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 them 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 whole 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 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 uppercase). Figure 3-11 shows examples of sorting all three lists in descending (reverse) order using `reverse=True`.

```
# Need this modules 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, because when you sort in reverse, you still actually sort: Z–A for strings, largest to smallest for numbers, 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 any way. The following code shows an

example in which we reverse the order of the names in the list and then print the list. The output shows the list items reversed from their original order:

```
# 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 ever need to work with a copy of a list, use the `.copy()` method so as not to alter the original list,. 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']
```

For future references, Table 3-1 summarizes the methods you've learned about so far in this chapter. As you will see in upcoming chapters, those same methods 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 within 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. Put `reverse=True` inside the parentheses to sort 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 like *tupple* (rhymes with *couple*). But it's not spelled *tupple* or *touple* so our best guess is that it's pronounced like *two-pull*. (Heck, for all we know, there may not even be a "correct" way to pronounce it. This probably doesn't stop people from arguing about the "correct" pronunciation at length, though.)

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 after it's defined you can't change it. 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 time an item appears within the 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 of the tuple contains 4.95. It returns False if it doesn't contain that.

If an item exists within 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 like –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 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 is as follows:

```
$29.95
$9.98
$4.95
$79.98
$2.95
```

You cannot change the value of an item in a tuple using this kind of syntax:

```
prices[1] = 234.56
```

If you try that, you'll get an error message that reads `TypeError: 'tuple' object does not support item assignment`. This 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 of the methods that alter data, or even just copy data, from a list cause an error when you try them with a tuple. This incudes `.append()`, `.clear()`, `.copy()`, `.extend()`, `.insert()`, `.pop()`, `.remove()`, `.reverse()`, and `.sort()`. 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 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 positions.

To define a set, use curly braces where you would have used 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 cannot get an item in a set based on its index number. Nor can you change an item that is already in the set.

You can't change the order of items in a set either. So you cannot 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)
```

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 shoe 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}
```

You can loop through a set and display its contents formatted with f-strings. The last couple of code lines in Figure 3-12 show a loop that uses `>6.2f` to format all the prices right-aligned within a width of six characters. The output from that code is shown at the bottom of the figure. You can see the output from the first few print statements. The list at the end is output from the loop that prints each value right-aligned.

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 very widely-used Python data structure is the data dictionary, which you will learn about in the next chapter.