O'REILLY®

AI-Assisted Python for Nonprogrammers

Learn Programming from the Ground Up

Reuven M. Lerner

AI-Assisted Python for Nonprogrammers

Learn Programming from the Ground Up

Reuven M. Lerner

AI-Assisted Python for Nonprogrammers

by Reuven M. Lerner

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

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

Acquisitions Editor: Michelle Smith

Development Editor: Corbin Collins

Production Editor: Aleeya Rahman

Copyeditor: Dwight Ramsey

Proofreader: Andrea Schein

Indexer: Krsta Technology Solutions

Cover Designer: Susan Brown

Cover Illustrator: José Marzan Jr.

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

October 2026: First Edition

Revision History for the First Edition

- 2026-09-18: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341661639* for release details.

979-8-341-66163-9

[LSI]

Dedication

For Shira, my favorite nonprogrammer since 1999.

Preface

For my 10th birthday, my parents bought me an unusual present: a class in computer programming.

The year was 1980, and it's safe to say that programming wasn't something that most people did, let alone most elementary-school students. And it's not as if my parents really knew much about computers, either. But they thought that I would enjoy it — and I did.

The class took place in the back room of a local computer store. Every week, the instructor would spend an hour or two demonstrating elementary programming using the Apple II computer at the front of the room. The students were, so far as I can remember, businesspeople in their 30s and 40s, who believed that computers were the way of the future, and that learning to program was the way to succeed in that future.

That class not only introduced me to programming, but got me hooked for life. There's something magical about our ability to type text into a machine, and then have the machine execute our instructions. Sure, programming can be frustrating, and even maddening — but there's almost nothing like the adrenaline rush you feel when a program works. And the fact that some of us can earn a living writing such code? Even better.

In the 40 years since I took that class, the tech industry has changed in numerous ways: computers have gotten dramatically faster, with far more memory and storage than we could have imagined. Computers are connected to one another, not just on local networks, but to the internet. Mobile devices put incredible computing power into our pockets, allowing us to communicate, learn, pay, and travel with ease. And in just the last few years, artificial intelligence has gone from a research project to a trillion-dollar industry, with repercussions for business and society that we cannot yet predict.

One thing that hasn't changed, though, is the centrality of programming in all of this. Being able to code — specify, write, debug, and iterate — is still an in-demand skill. And it's one that everyone can learn.

On the one hand, the number of programming books, courses, videos, and tutorials is so overwhelming that it's easy to believe that anyone can learn to program. But then the class begins, and it's no wonder why you're frustrated: the instructor might think that they're giving an introductory course, but they're actually giving an introductory course *for people who already know how to program*.

Taking such a class can be quite frustrating. Here you are, all prepared to learn a new skill — only to discover that the supposedly introductory class isn't for newcomers at all.

Learning to program involves at least two different skills:

- You need to learn the concepts associated with programming, which exist across all programming languages. What are values? What are variables? What are functions? What are loops?
- At the same time, you also need to be learning the specifics of a particular programming language — C, Java, or (in the case of this book) Python.

Many so-called introductory classes teach the second of these, but ignore the first. Not surprisingly, many people don't finish such courses, saying that programming is too hard and meant for nerdy math geniuses.

Moreover, someone who is just getting started programming needs extra practice and hand-holding. Most instructors can't or won't spend the time that newcomers need, patiently helping them to work through the inevitable conceptual, syntactic, and logistical challenges.

I'm a professional Python trainer, in business since 1995. Many of the classes I teach are for experienced developers who want to learn Python, or for experienced Python developers who want to improve their skills. But one of my favorite classes to teach, "Python for nonprogrammers," was the

inspiration for this book. That's because I love seeing people getting excited, as I did back in 1980, at what we can instruct the computer to do. It isn't always easy, and it does require practice, but at the end of the day, programming can be exciting and fun.

Programming is also a form of literacy — what learning scientists like my PhD advisor Uri Wilensky have called "computational literacy," building on earlier work by Seymour Papert. Sure, you could become a full-time programmer. But if you write or speak well, then you don't have to become a professional author; you can just become a better communicator in your current job. In the same way, knowing how to program just makes you better at your current job, able to tell the computer how to automate tasks, communicate with other systems, and analyze data.

This book will introduce you to the world of programming, slowly but surely. You'll learn fundamental concepts. You'll also learn Python, one of the most popular programming languages in the world — and also one of the easiest to learn. When you finish this book, you'll be ready to use Python to solve some of your problems, or even to take more advanced Python classes.

One key to this book are the exercises: just as you cannot learn a foreign language by reading an introduction to its grammar, you cannot learn to program in Python by only reading this book. Part of the learning process involves trying, making mistakes, learning from those mistakes, and then doing better the next time. If you don't manage to solve an exercise on your own, and need my explanation to understand things, that's just fine! Many of these concepts take a long time to learn and absorb.

The second key to this book, though, is artificial intelligence. The AI revolution has been going on for several years already, with products like ChatGPT and Claude upending how we think about work, school, and even programming. Many people believe that AI means that no one needs to learn to code, because AI products can write programs for us. I have lots to say on that controversial subject in Chapter 15 — but there's no doubt that AI can help you to *learn* better, and more effectively.

This is exactly what new programmers need, and what their human instructors so often can't provide: AI provides us with an infinitely patient guide that can answer our questions and satisfy our curiosity. But beyond just answering questions, AI can also help to challenge our assumptions, reinforce important ideas, and crystallize our thinking. Each exercise in this book encourages you not just to solve a problem yourself, but to use AI to further enhance your understanding of the topic you just learned. These techniques are applicable beyond this book, meaning that even when you finish this book, you'll be able to use AI to enhance your understanding throughout your Python career.

Also: I've created a practice system that runs in your browser, without needing to install any external software. Not only does it let you write, edit, and run Python code, but it includes an AI tutor that gives you feedback and encouragement as you work toward a solution. Clicking on the appropriate link in each exercise will bring you to my practice system. I hope that it, along with repeated practice and thinking, will help you to internalize and understand Python even better.

I hope that this book will, first and foremost, show you that, yes, even you can learn to program, despite previous frustrating attempts to do so. Second, I hope that it'll give you insights into how software works, and how it is written — insights that you can apply to your job, even if you never actually write any code. And if you find programming so exciting and addictive that you want to take more classes, and try more projects? I couldn't agree more — and neither could my 10-year-old self.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

> Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

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

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/reuven/ai-assisted-python-for-nonprogrammers*. Interactive versions of the book's exercises can be found at *https://practice.lernerpython.com/classroom/ai-nonprogrammers*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*AI-Assisted Python for Nonprogrammers* by Reuven M. Lerner (O'Reilly). Copyright 2027 Reuven Lerner LLC, 979-8-341-66163-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

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

O'Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

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

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/ai-assisted-python-nonprogrammers*.

For news and information about our books and courses, visit *https://oreilly.com*.

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

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Writing a book might seem like a one-person activity, but that is far from the truth. But the support goes beyond the editors, important as they are! I'll start by thanking my wife, Shira Friedman, and my children Atara, Shikma,

and Amotz, who gave me the time and encouragement I needed to work on the book.

I'm also grateful to my friends and colleagues in the Python community, not only for creating an amazing language and ecosystem, but for a constant stream of new ideas, as well as feedback on my talks, teaching, and writing.

I also want to thank the tech reviewers, who suggested numerous corrections, improvements, and alternatives to my original draft. This book is significantly better thanks to their help. Thanks to Bill Lubanovic, Kerim Satirle, Kelly Schuster-Paredes, Rodrigo Girão Serrão, and Stephen Gruppetta.

Lastly, I would like to thank the thousands of people who have attended my courses over the 30+ years that I have taught Python. Their questions, feedback, and exercise solutions have forced me to think deeply about how people learn Python, and how to teach it better.

Chapter 1. First Steps with Python

You're reading this book because you want to program in Python. But what does it mean to write a program? What is a programming language, and what is Python? In this chapter, you'll learn not just what Python is, and how it fits into the overall world of programming and programming languages, but how to prepare your computer to write and edit the programs you'll learn to create in the rest of this book.

What Is a Computer?

The industrial revolution, which started in Britain in the 18th century, marked a major turning point in human history. Before that, creating a piece of fabric was a time- and labor-intensive activity, requiring many hours of skilled labor. The invention of mechanized looms not only made it possible to weave fabric more quickly than before, but also more accurately and more consistently.

It also reduced the number of people needed to work in such factories. Soon, people found ways to automate other jobs that had, for centuries, been the domain of specialized, skilled artisans. Hundreds of years later, we're still feeling the economic and social effects of that profound change in society. Our world of high-quality, inexpensive goods is possible because of this shift.

But the industrial revolution only covered manufactured goods. If you needed to perform many calculations — such as a government or large company might need for budgeting or taxes — then you were still using human labor. You wanted to calculate more quickly? You needed more employees, sitting at more desks, to handle the task.

The 20th century's digital revolution changed all that. The first computers were monstrously large, taking up gymnasiums, and ran on vacuum tubes. But they worked, allowing governments to perform calculations more quickly and accurately than people could. That said, the computers were still expensive to design, build, and maintain, so only the largest, wealthiest institutions could afford them.

This was especially true because each computer was designed to solve one specific problem. Changing the problem — or the solution — meant rewiring, and often rebuilding, the computer.

Fortunately, the technology was rapidly improving. Vacuum tubes made way for transistors, which were smaller and more reliable. They also generated far less heat, and thus required less air conditioning. Then came silicon-based semiconductors, which could contain many transistors, reducing a computer's size even more.

At the same time, engineers created general-purpose computers, ones that could solve a wide variety of problems. The physical computer (the *hardware*) stayed the same, but the instructions (the *software*) changed. Both the instructions and the data it processed were represented in *binary code*, meaning 1s and 0s. (It's common to think of 1 as "on" and 0 as "off," but actually they're just higher and lower levels of electricity.) Writing these instructions became known as *programming* a computer. Each binary digit became known as a *bit*, and a collection of eight bits became known as a *byte*.

So, what is a computer? In our modern world, it's a general-purpose machine that follows instructions to operate on data.

What Is a Programming Language?

Programmable computers were a major improvement over their predecessors. But programming in binary code was difficult and error-prone. To make it easier to instruct a computer, people invented *programming languages*, allowing you to write in something resembling

English, but which was then translated into 1s and 0s. Today, even the most modern computer can, at the most basic level, only process instructions and data in bits and bytes. When you use a programming language, such as Python, your code must be translated into that binary form before the computer can process it.

This translation is performed, somewhat ironically, using software. That is: a program reads your code, and produces a *binary executable* that your computer can run. There are two basic translation techniques, which are used to distinguish between languages:

- *Compiled* languages are translated all at once, before execution takes place. This is similar to translating a book from (say) English into Spanish, and then distributing the book to Spanish speakers. The original English version isn't available to readers of the translated version, not that they would need it most of the time anyway. Readers of the translated edition might need to wait for the initial translation to happen, but once the translation is done, it's as available as any native Spanish-language book. Compiled languages tend to execute quickly, in part because the compiler has time to analyze your program and optimize the executable it creates. They also require that neither the original *source code* nor the compiler be installed. C, C++, and Rust are well-known compiled languages, and they have a deserved reputation for speed and efficiency. Your operating system and word processor are likely written in a compiled language.
- *Interpreted* languages, by contrast, are translated on the fly, as the program executes. It's similar, in many ways, to hearing a speech in a foreign language through a simultaneous translator. The original speaker must be there, and there's a slight delay as the translation comes through. But if something changes — the speaker ad-libs a joke, or deviates from the originally prepared text — then the translator can adjust things in real time. Interpreted languages typically execute more slowly than compiled ones, and require that both the source code and the programming language be around

during the execution. The Unix shell and Windows batch files are common examples of interpreted languages.

There is also a third category of language, a hybrid between these two traditional ones that tries to keep the advantages of each. Such languages are compiled into an intermediate format, often known as *bytecodes*. The bytecodes are then run on your computer. Java, C# (and other .NET languages), and JavaScript are all examples of such languages. This is why, even if you're not a Java programmer, you might find yourself having to upgrade Java on your computer. That's the interpreter of compiled Java bytecodes. Your browser has its own version of bytecodes, known as *WebAssembly* (abbreviated to *WASM*); any language that can be compiled to WASM can thus run inside your browser. WASM isn't only for browsers, but it's most commonly used there, allowing you to install software just by pointing the browser at a URL.

There are thousands of programming languages, and they differ from each other in ways that go far beyond how they are translated into binary format. Some, like C, are low-level languages. In such languages, the programmer needs to think in ways closer to those low-level bits and bytes, explicitly asking for memory from the operating system and then freeing it after it's no longer needed. The advantage of such a language is that programs tend to run extremely fast. Other languages, like Java, compartmentalize data structures using a technique known as *object-oriented programming*. Still others, such as Matlab, are optimized for mathematical operations on vectors and matrices, crucial in many areas of science, engineering, and statistics.

If you speak more than one language, you know that there are certain concepts that are easier to express in some than in others. For example, it's easiest to discuss US politics in English, Jewish law in Hebrew, and Chinese philosophy in Chinese. You *can* use another language, but you'll find yourself using words and phrases that approximate your real meaning. In the same way, different programming languages are appropriate for different tasks. You can, in theory, use any language for any purpose — but some tasks will be harder in some languages than in others.

What Is Python?

Python was invented in the early 1990s by Guido van Rossum, a Dutch computer scientist who was researching programming languages, and particularly a language called ABC that was meant to make programming accessible to the masses. He wanted to create a new language that was easy to learn and use, with an emphasis on readability, but which could also be used to solve large, serious problems.

He released Python on the internet in February 1991, and several things immediately stood out:

- Python is a *high-level* language, letting you write in a style that is closer to English than to a computer's binary code.
- Python has automatic memory management, freeing you from having to request and free memory while running your program. When data is no longer needed or used, the language automatically returns it to the operating system.
- Python is a byte-compiled language, compiled to bytecodes (made up of low-level instructions called *opcodes*), and then interpreted. This means that running a Python program requires that the user have a copy of Python on hand.
- Python supports a variety of programming techniques and styles, from simple scripts to large-scale programs that incorporate a number of files. And if you want to write object-oriented code, you can do that, too.

Sounds pretty good, right? Even from its earliest days, Python had some very enthusiastic fans. Newcomers to programming liked the shallow learning curve, making it easier to learn than C or Java. Experienced developers liked the fact that they could do all of the sophisticated things they wanted, but with a minimalist, readable syntax. Everyone liked the huge library of third-party packages that could extend Python's capabilities. And as time went on, it became clear that while writing software is hard,

maintaining software is even harder — and that adopting a language optimized for readability and maintainability gave organizations a business advantage.

Python's popularity particularly exploded in the wake of three things:

- First, MIT adopted Python as the language for its introductory course for computer-science majors. MIT had previously taught the course using Scheme, a variant of the classic Lisp language. Scheme is elegant and powerful, but wasn't used in any commercial settings. The 2008 switch to Python made the course not only useful for introducing concepts of programming and computer science, but also as an introduction to a practical language popular in industry. Thousands of institutions followed MIT's lead, and in many cases adopted their Python curriculum. This meant that CS graduates all over the world knew and preferred Python, either when looking for jobs or when starting their own companies.
- Second, processing and analyzing data became an increasingly important part of the programming ecosystem. Python's facility with data was useful for programmers, who were happy to use a single language for a wide variety of tasks. But Python's shallow learning curve meant that data analysts without a strong programming background could learn the language and then apply it to their work with relative ease. This led to a virtuous circle: more people used Python for data, which led to rapid improvements in Python's data-processing functionality. That attracted even more people to Python, and the cycle continued.
- The third reason is the widespread adoption of Python 3, first released in 2008. Python 3 was a major upgrade to the language, providing facilities that would make it more consistent, easier to develop, and suitable for larger and more sophisticated programming projects. But this came at a cost, namely compatibility with Python 2. For this reason, the upgrade to Python

3 caused enormous division and confusion within the Python community, and took many years to become mainstream. It wasn't always obvious that Python 3 would win the day, or that Python would survive this transition. But it did — and Python 3 increasingly demonstrated its superiority over Python 2. Moreover, Python 3 was (and is) more than a simple language appropriate for simple tasks; it's a first-class programming language, one that allows coders to solve even the stickiest of problems.

The Python Community

You could argue that there's a fourth reason as well: an organized community dedicated to making Python better for all of its users, from complete newbies to experienced professionals.

For too long, programming was seen as too difficult for most people to learn. It was for experts, not for the masses. The rise of so-called "scripting" languages demonstrated that programming, while a skill that requires study and practice, was within most people's reach. But the other languages (e.g., Tcl, Perl, and Ruby) have a small fraction of Python's user base.

It wasn't an accident that Python has outlived and surpassed its competition. Guido van Rossum wanted Python to be welcoming to newcomers while remaining a viable option for serious developers with complex needs. Seymour Papert, who invented the Logo programming language for children, called this approach "low floors and high ceilings," and you could argue that Python deserves this mantle more than Logo ever did.

Moreover, van Rossum decided early on to formalize the role of the Python community. Between the core developers (who write Python, and have established standards for who can contribute), the Python Software Foundation (a nonprofit which owns the Python trademark and copyright, and which provides a support structure for core developers), Python Enhancement Proposals (to standardize the process for adding new language features), and a global network of conferences (starting with PyCon in the US and EuroPython in Europe), Python both offered

numerous opportunities for people to contribute and also for them to feel a part of the language's community.

In many ways, this social infrastructure, no less than the technical infrastructure, helped the language to expand into numerous domains and to attract new users. And indeed, Python can today be found at organizations of every size, used by programmers of every background, and used in a wide variety of tasks. Python is particularly popular in:

Data

Whether it's data science, data analytics, or machine learning, Python plays a prominent role in anything having to do with data.

APIs

Computers communicate with one another via specified protocols known as *application programming interfaces*, or APIs. API services are often written in Python, and the programs that consume such APIs frequently are, as well.

Web applications

Newspapers, financial institutions, ecommerce shops, and many other companies' websites, both internal and external, are powered by Python.

Automated testing

Software and hardware companies often use Python programs to run millions of tests while products are under development, and then before they're shipped off to customers.

Devops

Companies now need to configure, deploy, and coordinate thousands of servers, often in response to customer demand — and Python is often used to configure and control these rollouts.

Education

> Not just universities, but high schools, middle schools, and professional enrichment courses are all teaching Python, all over the world.

There are places where Python isn't popular, or even appropriate: if a program must run at high speed, or use as little memory as possible, then Python is a bad choice. You thus won't see operating systems, browsers, or database engines written in Python. Also, while it's possible to write mobile applications using Python, support for doing so is still somewhat new, although it is improving quickly.

JupyterLite

If you want to write and execute Python code, then you'll need to install Python on your computer. That installation has gotten easier over the years, but it can still be a bit tricky, especially if you're new to programming.

Fortunately, there is a workaround in the form of WebAssembly (WASM), which allows Python to run inside your web browser. Installing the WASM version of Python, known as Pyodide, is as easy as pointing your browser at a website. Pyodide uses the WASM technology that you read about earlier.

However, it's not enough to just download and install Python. You also need a way to write and edit code, as well as execute it. That's where JupyterLite comes in — it makes it easy to jump right in and start writing code.

A bit of background: for more than a decade, the Jupyter project has provided a browser-based way to write Python code. It was particularly popular among data analysts and data scientists, since Jupyter's notebook paradigm allows data, documentation, code, and visualizations to coexist in the same document.

However, Jupyter has traditionally required that you install Python and the Jupyter package itself, both of which can be a bit tricky for first-time users to handle. Plus, not all organizations allow employees to install new programming languages, meaning that there can be some bureaucracy and security negotiations involved.

JupyterLite is more than enough for solving the exercises in this book. You may, at some point, decide that JupyterLite isn't enough, and that you want to move up to the full-fledged version of Jupyter. Or you might want to move to a Python editor, such as PyCharm or VS Code. That's more than reasonable, and you can do so by following the instructions later in this chapter. But for ease of installation, and being able to rapidly get started, nothing beats JupyterLite.

So, how do you get started with JupyterLite? First, go to the JupyterLite home page, shown in Figure 1-1. On the left, you will see a short list of files and folders. These are the files that JupyterLite can see. They aren't really on your disk, but rather stored inside of your browser. However, you can drag files from your computer's disk to this left-hand panel if you want JupyterLite to see them.

The main pane of the JupyterLite startup page contains colored icons representing different functionality you can try. To start up in Python, you'll want to select the Pyodide/WASM box, typically in the top left. Click on that, and you'll see a Jupyter notebook document open, as shown in Figure 1-2.

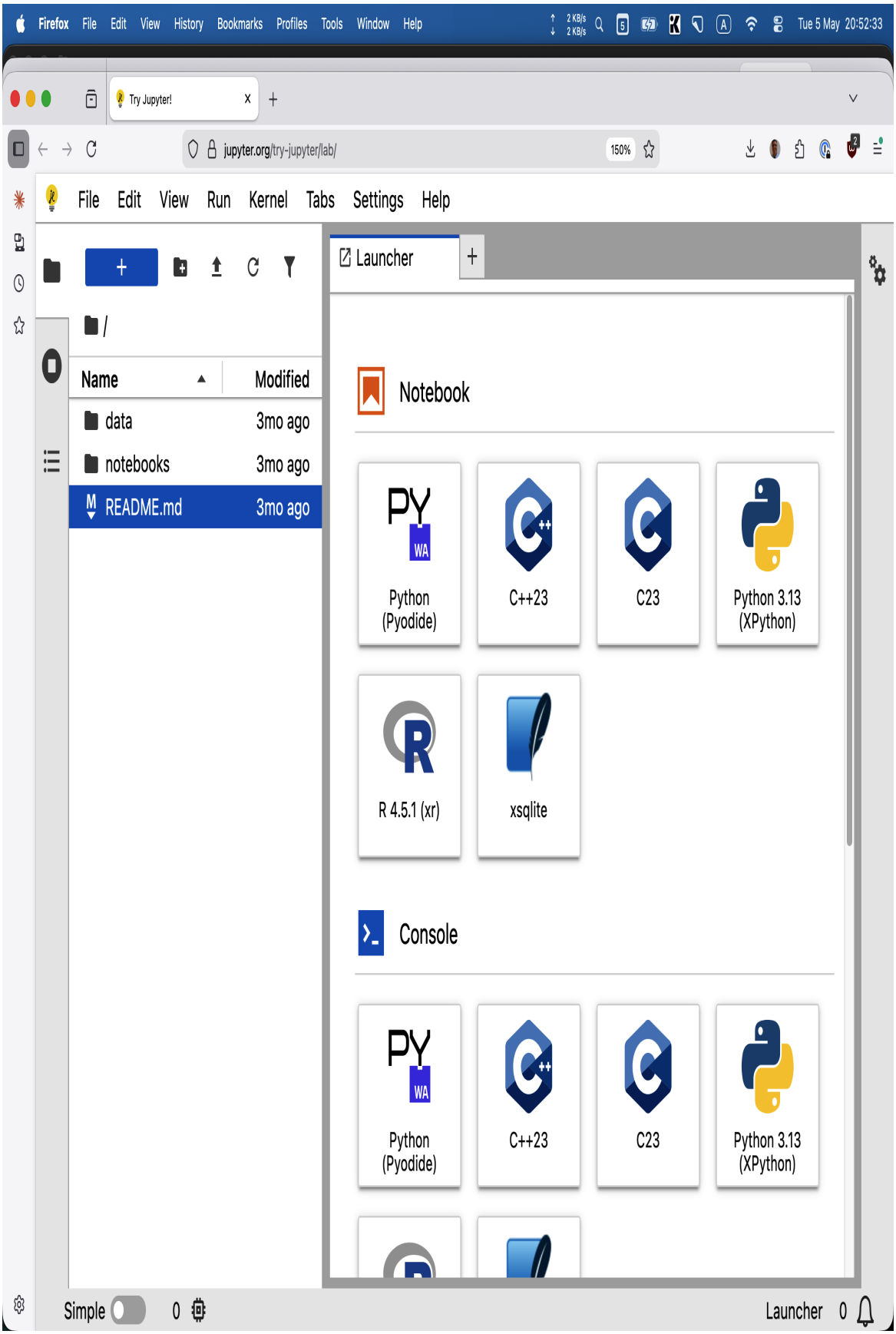

Firefox File Edit View History Bookmarks Profiles Tools Window Help
Tue 5 May 20:52:33
Try Jupyter!
jupyter.org/try-jupyter/lab/
150%
File Edit View Run Kernel Tabs Settings Help
Launcher
/
Name
Modified
data 3mo ago
notebooks 3mo ago
README.md 3mo ago
Notebook
Python (Pyodide)
C++23
C23
Python 3.13 (XPython)
R 4.5.1 (xr)
xsqlite
Console
Python (Pyodide)
C++23
C23
Python 3.13 (XPython)
Simple
0
Launcher
0

Figure 1-1. JupyterLite startup page

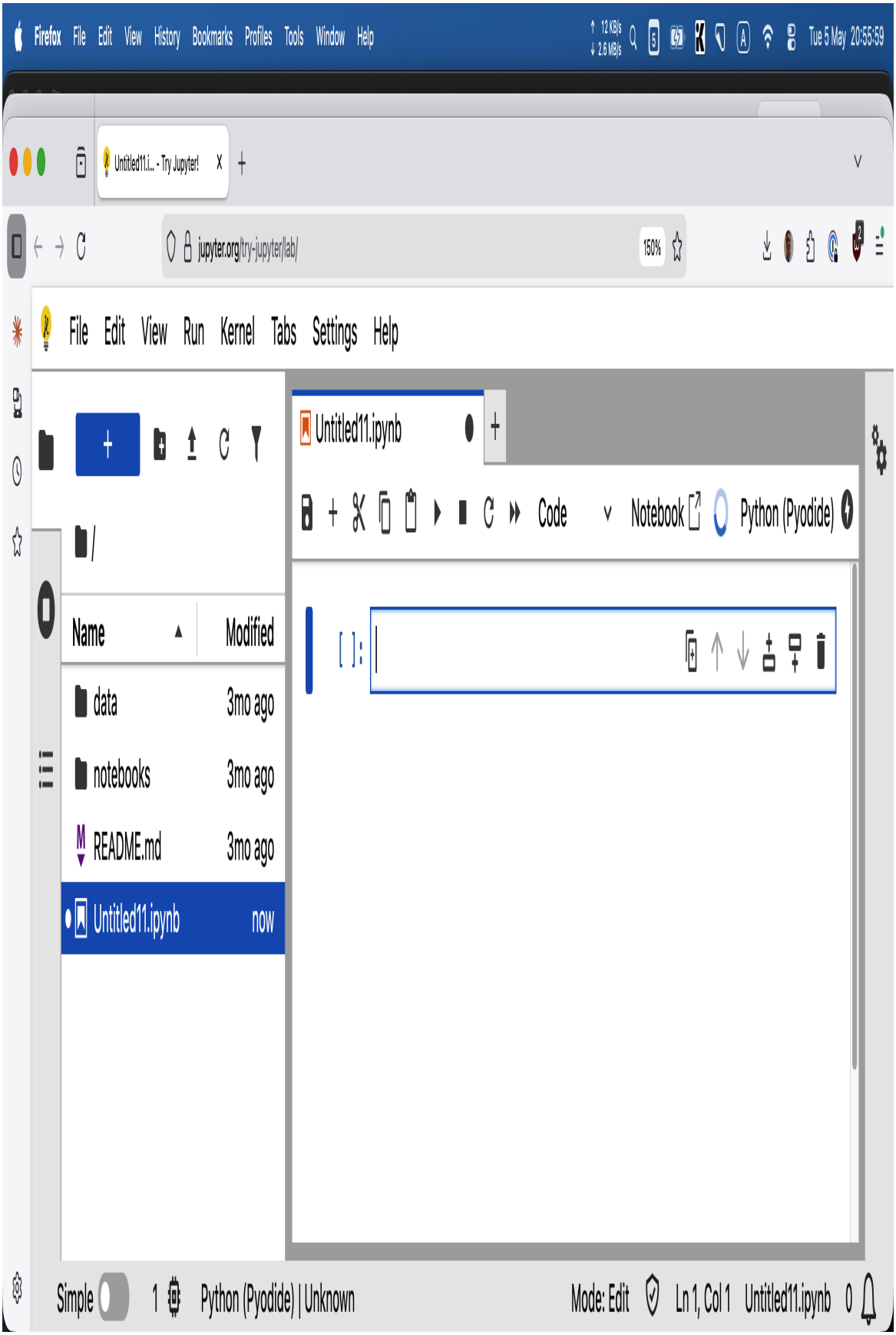

Firefox File Edit View History Bookmarks Profiles Tools Window Help
Tue 5 May 20:55:59
Untitled11.i... - Try Jupyter!
jupyter.org/try-jupyter/lab/
150%
File Edit View Run Kernel Tabs Settings Help
Untitled11.ipynb
Code
Notebook
Python (Pyodide)
Name
Modified
data 3mo ago
notebooks 3mo ago
README.md 3mo ago
Untitled11.ipynb now
Simple
1
Python (Pyodide) | Unknown
Mode: Edit
Ln 1, Col 1
Untitled11.ipynb
0

Figure 1-2. JupyterLite starting a new notebook

Now what? You can start to write and execute Python code. Click inside the outlined rectangle, which the Jupyter world calls a *cell*. In that cell, write the following Python code:

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

You'll probably see that the text is colorized, with the function `print` in one color and the text `'Hello, world'` in a different color. You'll learn more about functions and text in Chapter 2, so don't let the specifics bother you. But having different types of words in different colors makes it easier to read someone else's code.

You can then execute the code either by pressing Shift+Enter together, or by pressing the *play* button (i.e., a black triangle). Either way, you should then see `'Hello, world'` printed on the screen, as in Figure 1-3.

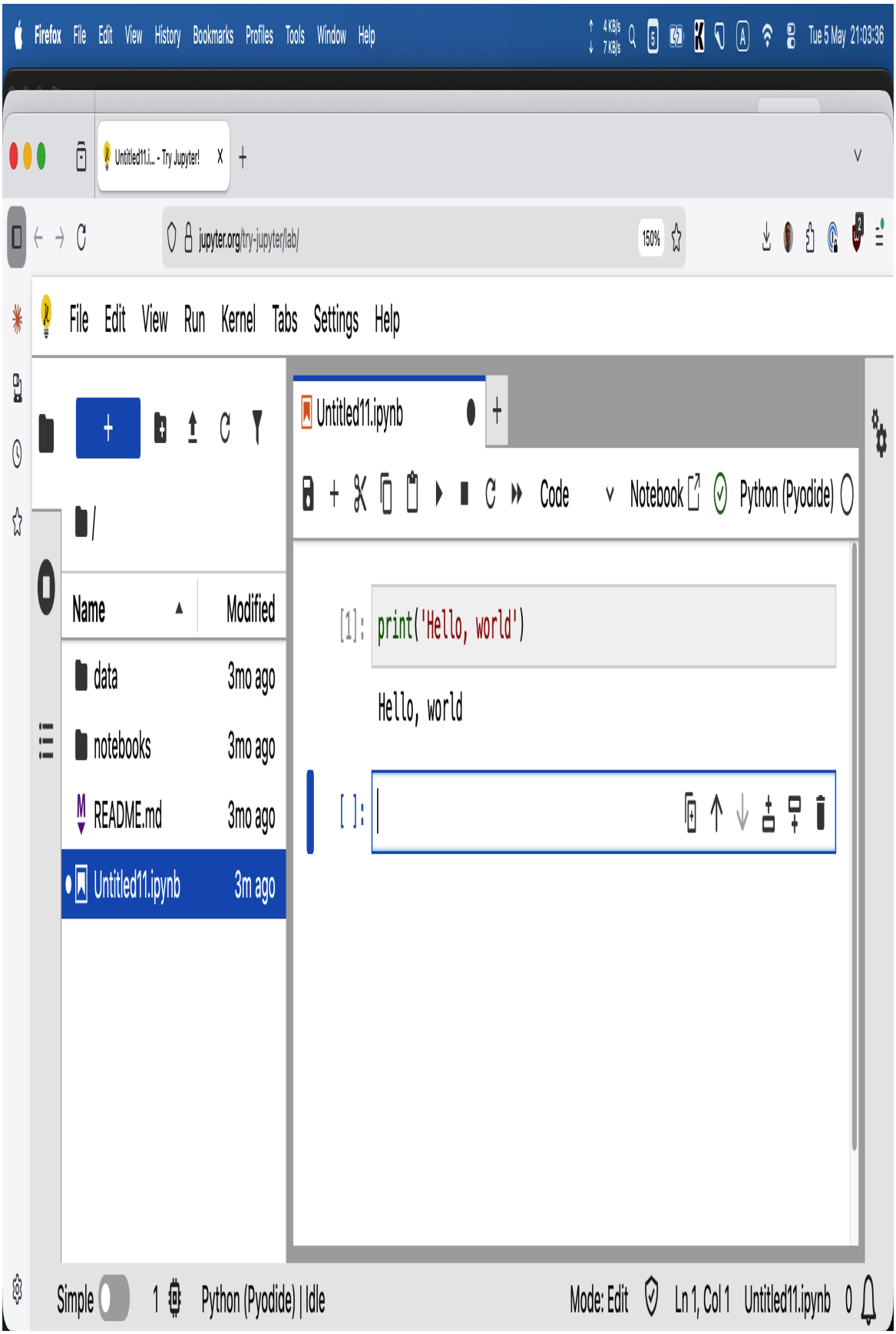

Firefox File Edit View History Bookmarks Profiles Tools Window Help
Tue 5 May 21:03:36
Untitled11.i... - Try Jupyter!
jupyter.org/try-jupyter/lab/
150%
File Edit View Run Kernel Tabs Settings Help
Untitled11.ipynb
Code
Notebook
Python (Pyodide)
/
Name
Modified
data 3mo ago
notebooks 3mo ago
README.md 3mo ago
Untitled11.ipynb 3m ago
[1]: print('Hello, world')
Hello, world
[ ]:
Simple
1
Python (Pyodide) | Idle
Mode: Edit
Ln 1, Col 1
Untitled11.ipynb
0

Figure 1-3. After printing "Hello, world"

You can do lots of other things in Jupyter, too:

- You're currently in *edit mode*, which means that everything you type is put into the current cell. Enter edit mode by clicking in the cell or pressing Enter.
- You can also be in *command mode*, where you typically type one character at a time, instructing Jupyter how to behave. Enter command mode by clicking to the left of a cell or pressing Escape. Among the command-mode commands you can use:
  - `c`, to copy the current cell
  - `x`, to cut the current cell
  - `v`, to paste the most recently cut or copied cell into the current location
  - `a`, to add a new, empty cell *above* the current one
  - `b`, to add a new, empty cell *beneath* the current one
  - `y`, to indicate the current cell contains Python code
  - `m`, to indicate that the current cell contains GitHub-flavored Markdown text

Jupyter is a modern incarnation of a classic programmer tool known as a *REPL*, short for "read-eval-print loop." Whatever you type into the REPL is read and evaluated by the programming language. The result is then printed on the screen — after which you can enter something else. Python comes with a command-line REPL sometimes known as the *Python interactive shell*, but it only runs in a terminal window.

Generations of programmers have enjoyed using REPLs because they allow for an interactive, exploratory form of programming. This is especially

useful in data analysis and education, so it's no surprise that Jupyter has become popular in those domains.

One particularly nice Jupyter feature is that it displays the value of a Python expression, even if you don't explicitly ask to print it. That is, you can type the following into Jupyter:

```
print(2 + 3)
```

Not surprisingly, you'll see `5` printed on the screen. But you can also write:

```
2 + 3
```

without any invocation of `print`, and you'll see `5` on the screen. Being able to see the value of any Python expression is especially useful once you start defining variables, since you can just ask to see a value by entering the name of a variable in a cell by itself.

Note that a JupyterLite notebook is stored inside your browser. This means that for someone else to see the notebook, you'll need to export it (using the Download option under the File menu) and then send it to them.

Jupyter, like all tools, takes some getting used to in order to take full advantage of its functionality. But it's worth learning, and is more than adequate for solving the problems in this book, as well as exploring Python down the road.

Installing Python

JupyterLite is enough for everything in this book. If you'd like to go further — working with files, or the AI agents in Chapter 15 — you'll need to install Python locally. Feel free to skip this and come back later.

Go to the Python home page, shown in Figure 1-4.


Firefox File Edit View History Bookmarks Profiles Tools Window Help
Wed 6 May 22:08:39
Download Python | Python.org
www.python.org/downloads/
Python
PSF
Docs
PyPI
Jobs
Community
python
Donate
Search
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
Download the latest version for macOS
Download Python 3.14.4
Looking for Python with a different OS? Python for Windows, Linux/Unix, macOS, Android, other
Want to help test development versions of Python 3.15? Pre-releases, Docker images

Figure 1-4. Python home page

If it guesses your computer correctly, you can just click on the Download button on the page. You can, via the provided links, instead download for another type of computer. The download will give you the installer that is appropriate for your computer and operating system; follow the instructions to make sure that it's installed.

> **NOTE**
>
> If you're using Windows, then make sure to select the checkbox that adds Python to your `PATH`. This makes it easier for you to find and invoke Python.

Once you have installed Python, open a new terminal window. You should then either type:

```
python3
```

or (on Windows):

```
py -3
```

If all goes well, then you should see, inside the terminal window, the Python interactive shell, the older, text-based REPL that helped to inspire the development of Jupyter:

```
Python 3.14.4 (main, Apr  7 2026, 13:13:20)
    [Clang 21.0.0 (clang-2100.0.123.102)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

When it starts, Python tells you what version is running (3.14.4, in this case) and when it was created. It'll also give some terse hints about what you can type for more information about Python.

Then, it prints the most important thing, the >>> prompt indicating that you can type some Python code into it. Since this is a REPL, you can just type:

```
2 + 3
```

and you should see `5` on the next line after you press Enter.

If this works, then congratulations! You now have Python installed and running on your computer. You can exit from the REPL by typing `quit` and pressing Enter.

You can then try to install Jupyter. This is done using Python's `pip` command, about which you'll learn more in Chapter 13. For now, type the following on the command line, *not* within Python itself. If you see the >>> prompt, then you should exit before typing this command:

```
python3 -m pip install jupyter
```

Or, if you used `py -3` to start Python before, you should instead write:

```
py -3 -m pip install jupyter
```

If all goes well, this should install the Jupyter software onto your computer. (It'll likely take some time, and will also print all sorts of status updates that won't particularly interest you.) You can then type:

```
python3 -m jupyter notebook
```

Or if you used `py -3` to start Python, use:

```
py -3 -m jupyter notebook
```

This should, after a bit of churning and thought, open Jupyter in your web browser, with a list of the files in your current directory, as shown in Figure 1-5.

You can create a new notebook by clicking on New and then selecting Python 3 from that list. Depending on your computer's configuration, you'll see a variety of other options there, as well; Figure 1-6 shows the menu.

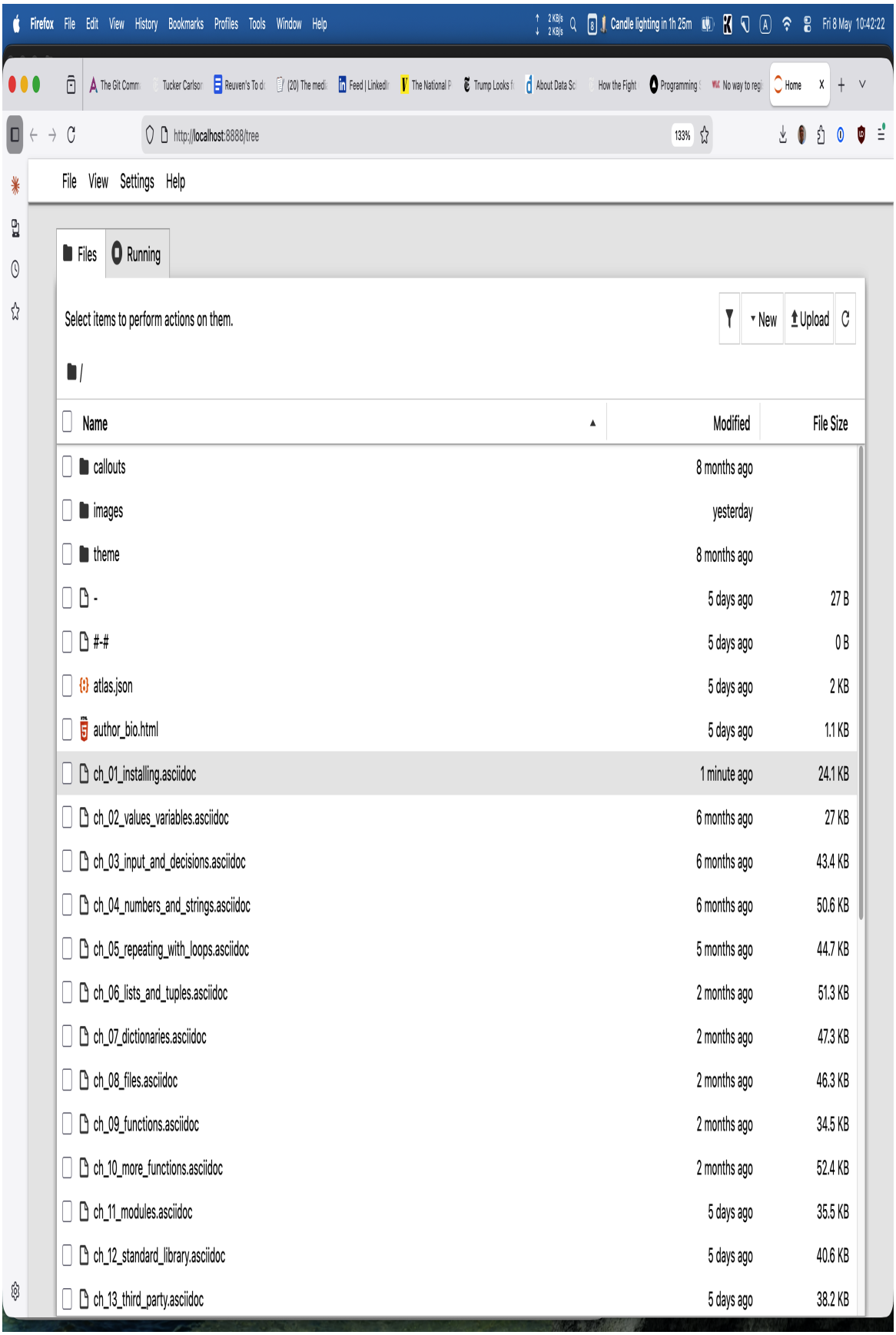

Firefox File Edit View History Bookmarks Profiles Tools Window Help
Candle lighting in 1h 25m
Fri 8 May 10:42:22
Home
http://localhost:8888/tree
133%
File View Settings Help
Files
Running
Select items to perform actions on them.
New
Upload
/
Name
Modified
File Size
callouts 8 months ago
images yesterday
theme 8 months ago
- 5 days ago 27 B
#-# 5 days ago 0 B
atlas.json 5 days ago 2 KB
author_bio.html 5 days ago 1.1 KB
ch_01_installing.asciidoc 1 minute ago 24.1 KB
ch_02_values_variables.asciidoc 6 months ago 27 KB
ch_03_input_and_decisions.asciidoc 6 months ago 43.4 KB
ch_04_numbers_and_strings.asciidoc 6 months ago 50.6 KB
ch_05_repeating_with_loops.asciidoc 5 months ago 44.7 KB
ch_06_lists_and_tuples.asciidoc 2 months ago 51.3 KB
ch_07_dictionaries.asciidoc 2 months ago 47.3 KB
ch_08_files.asciidoc 2 months ago 46.3 KB
ch_09_functions.asciidoc 2 months ago 34.5 KB
ch_10_more_functions.asciidoc 2 months ago 52.4 KB
ch_11_modules.asciidoc 5 days ago 35.5 KB
ch_12_standard_library.asciidoc 5 days ago 40.6 KB
ch_13_third_party.asciidoc 5 days ago 38.2 KB

Figure 1-5. Jupyter startup page

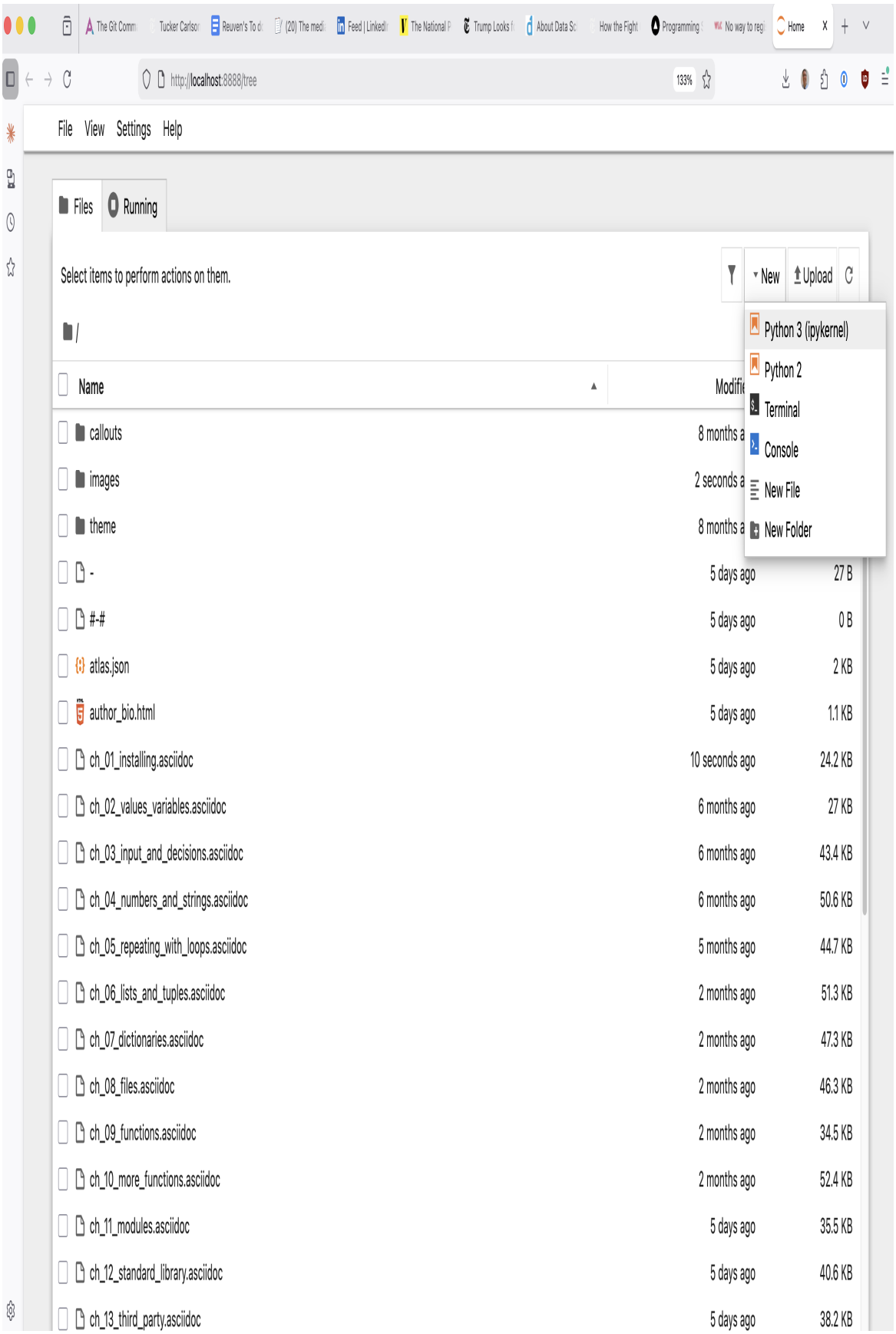

Home
http://localhost:8888/tree
133%
File View Settings Help
Files
Running
Select items to perform actions on them.
New
Upload
/
Python 3 (ipykernel)
Python 2
Terminal
Console
New File
New Folder
Name
callouts
images
theme
-
#-#
atlas.json
author_bio.html
ch_01_installing.asciidoc
ch_02_values_variables.asciidoc
ch_03_input_and_decisions.asciidoc
ch_04_numbers_and_strings.asciidoc
ch_05_repeating_with_loops.asciidoc
ch_06_lists_and_tuples.asciidoc
ch_07_dictionaries.asciidoc
ch_08_files.asciidoc
ch_09_functions.asciidoc
ch_10_more_functions.asciidoc
ch_11_modules.asciidoc
ch_12_standard_library.asciidoc
ch_13_third_party.asciidoc

Figure 1-6. Creating a new notebook

Once you create the new notebook, it'll work much like what you did with JupyterLite. The big difference is that Python is running on your computer, rather than just in the browser. As such, it can read and write files on your computer, and there is no need for a private list of files, which you had previously seen in JupyterLite.

Moreover, the notebook file will be saved on your computer, in the directory where you initially started Jupyter. If you want to load, run, or modify this notebook in the future, just start Jupyter in the same directory where you were before, and select the notebook file from the list of files in the folder. You can also copy or move the file to another directory, or even share it with a colleague.

IDEs

Notebooks are one of the newest and most interesting recent developments for people working with code. They are perfect for an environment in which data, code, documentation, and visualizations need to exist in the same place, and where you're constantly experimenting with code.

> **NOTE**
>
> You don't need to install an IDE to get started with Python. Feel free to wait until you first have gained some experiernce using JupyterLite. Then you can come back, and install both Python and an IDE with a bit more knowledge and confidence.

But notebooks aren't appropriate for all Python projects. If you're developing a large application, then you'll be writing a number of different files, and might well be collaborating with colleagues. In such a case, you'll need an editor that will help you write and debug your Python files, pointing to potential code improvements and using the Python world's latest tools for improving code quality.

For many years, programmers used *code editors* for such purposes. These editors, including programs like Emacs and vim, are still popular among many coders, especially more veteran ones. But most programmers today use an integrated development environment, or *IDE*, such as PyCharm or VS Code. IDEs are more than just code editors. They provide all of the infrastructure that you'll need, from code autocomplete to AI integration to debugging to package management.

Programmers tend to be somewhat fanatical about what coding editor they use. Try several of them (not immediately, and not all at once), to see which one suits you best! You'll likely find that some editors reflect your conventions and workflow better than others. Two of the most popular editors are PyCharm (which also has a paid upgrade with more functionality) from JetBrains and VS Code from Microsoft.

PyCharm is a Python-specific version of an editor that JetBrains customizes for many different programming languages and environments. VS Code, by contrast, is a general editor, and thus requires the installation of a Python plugin in order to handle Python correctly. Both support numerous plugins, connections to AI systems, and version control. Both support the creation and editing of Jupyter notebooks directly within the editor itself, without installing or launching a separate program for Jupyter. And both have extensive documentation and help, to help you get the most out of each system.

You don't need to install an IDE to learn Python or to use it, and certainly not to get through this book. But at some point, it's worth exploring the use of an IDE, if only to get experience working with a more sophisticated tool with greater support for large, complex code bases.

Artificial Intelligence

So far, you've learned how to get Python up and running on your computer. But this book isn't just about Python. It's also about using AI to help you program and use Python. Where's the AI?

For the most part, the AI in this book is incorporated into the exercises. You'll not only have to solve programming problems, but also use AI to reinforce what you learned in solving the exercises. In Chapter 15, you'll also see how AI can help you to write code, but the emphasis in this book is on using AI to learn better, not to code faster.

In order to do the AI challenges, you'll need access to an AI chatbot, such as Gemini, ChatGPT, or Claude. Any of them, even on a free tier, will be more than good enough for this book. The example AI interactions will rotate among ChatGPT, Claude, and Gemini, but you should have similar results using another GenAI system, if you prefer.

Summary

In this chapter, you learned why we need programming languages, why Python has become so popular, and some of the domains in which it is popular. You also learned how to use JupyterLite in your browser, how to install Python and Jupyter on your computer, and even about why you might want to use an IDE to develop Python code. You're now set to take your first steps coding in Python, which we'll begin in Chapter 2, looking at values and how we can store and retrieve them.

Chapter 2. Values and Variables

In many ways, programming languages are like human languages. For starters, human languages have verbs, which describe our actions: I *cook*, you *drive*, they *write*. The equivalent in a programming language is a *function*, which performs its actions when you "call," "invoke," or "execute" it. Every programming language comes with some built-in functions and invites you to define your own (as you'll see in Chapter 9). One of the most common functions, which you'll soon use to write a message on the screen, is `print`.

In a human language, you use a verb with a noun: I cook *spaghetti*, you drive a *car*, and they write a *letter*. In a programming language, these nouns are known as *values* or *objects*. And just as we can divide the world into many types of nouns, we can divide Python values into different *types*. Some types store numbers, others store text, and still others are *container* types, which hold other values. Every programming language comes with a number of built-in types. Python comes with a particularly large, rich set of built-in types, and even allows you to create your own.

It's important to remember that the computer doesn't know anything about these numbers, text, or containers; these are constructs that exist in a high-level programming language and are translated into 1s and 0s by Python. That's the beauty of a high-level language, and the reason why it's referred to as "high level": You can think in bigger, more human-like terms, what computer scientists often call a *higher level of abstraction*. The bits are always there, but unless something goes terribly wrong, you can usually ignore them in favor of higher-level, more human-like thinking.

It's similar to knowing that everything in the world is made up of atoms — but you wouldn't want to build a bridge or cook breakfast at the atomic level.

This chapter introduces some of the most important, basic types of values in Python, as well as how you can store them for later use. By the end, you'll know how to create them, use them, perform some basic operations on them, and even store them for later use in your program.

Displaying Values with print

In the last chapter, you set up your Python environment and had it print the traditional "Hello, world" message on the screen:

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

Let's point out a few things about the preceding code:

- To invoke `print`, or any function, put round parentheses, `()`, after its name.
- Inside the parentheses, you can pass zero or more *arguments*, values that you want the `print` function to use.
- In this case, the value is text, known to programmers as a *string*. Python knows that it's a string because of the quotes around the text.
- You can use either single (`''`) or double (`""`) quotes around text; from Python's perspective, it makes no difference.

When you execute this code, you see this text:

```
Hello, world
```

Note that the quotes are not printed; they were there to tell Python that `'Hello, world'` should be treated as a string. But when you display values to the user, the quotes are not shown.

Given how often you'll want programs to display things on the screen, you'll find yourself invoking it often. But you can pass just about any

Python value to `print`, not just text. This includes numbers, about which you'll learn in greater detail in Chapter 4. But for now, you can say:

```
print(10)
```

Not surprisingly, Python prints `10` on the screen.

What if you want to use our computer as a very expensive calculator? Does it know how to add things together? Yes, it does:

```
print(10 + 3)
```

The preceding code, when executed, will print `13` on the screen.

But let's stop for a moment and consider what just happened here:

- You took two numeric values, `10` and `3`, and used the `+` symbol, known as the *addition operator*, on them.
- When you add two numbers together, you get a new number — in this case, `13`.
- When you invoke a function with `()`, anything inside the parentheses gets run first. So before `print` ever starts to run, Python has already calculated `10+3` and replaced it with `13`.
- `print` receives the argument `13`, and prints it on the screen.

EXPRESSIONS

If `10` is a Python value, and `20` is a Python value, then what is `10 + 20`? Python calls it an *expression*, a value or instructions that result in a value.

Why do you need to know this? Because there are numerous places in Python that expect to see an expression.

For example, consider `print`. You can say `print(30)`. Or you can say `print(10 + 20)`. In the first case, the expression `30` is a value that cannot be simplified any further. So it invokes `print` with that value of `30`. In the second case, Python sees an expression that can be simplified. So it adds `10` and `20` together, gets the value `30`, and then invokes `print` with the result.

Anywhere that Python expects a value, you can provide an expression instead. If it is easier for you to write something in a longer form, as an expression, rather than as a single value, feel free to do that.

Operations and Text

Can you also perform operations on strings? For example, can you use + to combine two of them? Let's see:

```
print('Hello,' + 'there')
```

It turns out that yes, + works with text. You get back a value, a new string. And because you handed that expression to `print`, the computer prints something on the screen:

```
Hello,there
```

Now, this isn't *wrong*, per se. But it seems a bit weird that there isn't a space between the two words.

The thing is, this isn't strange to most programmers. That's because computers don't do what you *want* them to do, but rather what you *tell* them to do. Since you didn't tell Python to put a space between the words, it didn't put one there.

It's true, + allows you to combine two strings together. The result is a new string, one combining the values that you gave to +. But the new string will contain precisely the characters from the two input strings, no more and no less — and that includes space characters.

To get it to print a bit more nicely, you have to put in an explicit space character:

```
print('Hello, ' + 'there')
```

(I chose to put the space at the end of the first string. I could have also put it at the start of the second string.)

This time around, you get the result you wanted:

```
Hello, there
```

We can even use + more than once, combining multiple strings:

```
print('Hello, ' + 'there' + '!')
```

Sure enough, it prints the full sentence:

```
Hello, there!
```

Numbers Versus Digits

We've seen that you can add together two numbers:

```
print(10 + 3)
```

We've also seen that you can add together two strings, known as *concatenation*:

```
print('Hello, ' + 'there!')
```

What if the strings contain only digits? That is:

```
print('10' + '20')
```

As far as Python is concerned, you are applying + to two strings. This means that you'll get a new string back, the result of combining `'10'` and `'20'`.

> **NOTE**
>
> A string, from Python's perspective, is anything between quotes. When you see quotes around characters, Python sees it as a string:
>
> - `123` is not a string, but an integer.
> - `'123'` is a string.
> - `print` is not a string, but a function name.
> - `'print'` is a string containing the text `'print'`.

Sure enough, the preceding code prints the following:

```
1020
```

As humans, this seems odd. But this just demonstrates that integers (whole numbers, known in the Python world as `int`) and strings (text) are two completely different types of data, and thus behave differently. Digits without any quotes around them are seen as numeric data, and + works on them accordingly. Text, with any combination of letters, numbers, symbols, and even spaces, is always treated the same way by +, namely by tacking the second one onto the first. In the next chapter, you'll start to see how you

can convert values from one type to another. But for now, just remember that there's a world of difference between `10 + 3` (i.e., adding two numbers) and `'10' + '3'` (adding two strings).

Mixing Different Kinds of Values

We know that `10 + 3` gives you the integer `13`, and that `'10' + '3'` gives you the string `'103'`. What happens if you try to add an integer and a string? That is, what if you say:

```
print('10' + 3)
```

There are basically three possible outcomes for this code:

- Python notices that `'10'` is a string, but contains only digits. So it converts the string `'10'` to the integer `10`, and the result is the integer `13`, which is printed on the screen.
- Python sees that we're trying to apply `+` to a string and an integer. So it converts the integer `3` to a string, `'3'`, meaning that you're really adding `'10' + '3'`, and you get back `'103'`.
- Python realizes that it can't know your intentions, and it refuses to guess, giving an error message.

If you're designing a programming language, then you can decide how the language should resolve such issues. I've used languages that used each of the first two rules — which seemed great and intuitive, until it didn't quite work for me, because the language guessed wrong at least once.

Python takes the third approach, giving an error message (known as an *exception*) if you ask it to combine two values for which there isn't any clear answer:

```
print('10' + 3)
```

We get:

```
TypeError: can only concatenate str (not "int") to str
```

These errors are frustrating when you start to program, especially because the messages are often less than helpful to beginners. But over time, you'll likely grow to be grateful that you get such messages, and that they tell you the nature of the problem. It's a much bigger issue if your program doesn't work correctly, but also doesn't raise any exceptions indicating that there was an obviously identifiable problem.

In this case, Python is telling you that you can only concatenate (i.e., join together) a string value to another string value. You cannot do that with an integer. Which is, of course, precisely the problem we're experiencing here.

Variables and the Assignment Operator, =

So far, you have worked with *literal* integer and string values, meaning that you typed them each time you needed them. There's nothing technically wrong with this; in theory, you could write an entire program this way.

But doing so would be quite tedious. For one, you would have to type the entire value each time you want to use it — which isn't so bad if the values are `'hello'` and `10`, but gets more annoying as the values grow larger and more complicated.

Fortunately, programming languages allow you to store a value in a name. Such a name is known as a *variable*. If values are the nouns of a programming language, then variables are the pronouns — taking the place of a value, such that you can refer to it in a shorter, easier way.

Imagine if human languages didn't have pronouns, that instead of saying *he*, *she*, *they*, or any other pronoun, you would have to use the person's full name each time you wanted to refer to them. Pronouns make language shorter and more flexible. In the same way, variables make your programs shorter and more flexible.

We can assign a value to a variable with =, known as the *assignment operator*. For example, to assign the integer `5` to the variable `x`, you can say:

```
x = 5
```

Notice a few things here:

- Yes, we're using =, which in mathematics is known as the "equals sign," and means that the value on the left is the same as the value on the right. In Python, that is *not at all* what = means. Rather, it's an instruction to the computer, telling it to take whatever value is on the right and assign it to the variable on the left.
- The right side is evaluated before the left side, and can contain any Python expression. So it can be as simple as `5`, or something more complex, such as `5 + 100`. The value from that expression, whatever it is, is assigned to the variable on the left.
- If `x` already existed, then it now refers to `5`.
- If `x` didn't exist before, then it is created, and refers to `5`.

In other words: After running that line of assignment, you can be sure that `x` exists and now refers to `5`, regardless of whether it existed before or to what value it was referring.

So what?

Now you can use `x` in place of the integer `5`. For example, `print(x+x)` will display 10 on the screen. You can use `x` wherever you used to use `5`, such as in adding `x+x+x`, giving you `15`. This association between `x` and `5` will remain either until the program ends or you assign a different value to `x`.

Naming Variables

I've used `x` as a variable name here, and Python doesn't mind. But Python isn't the real audience for your variable names. Rather, it's the people who will be reading and maintaining your code after it is written — which means not only your colleagues, but also yourself. I know from bitter experience that when I maintain my own code, its meaning is far less clear than when I originally wrote it. Good variable names make it easier to understand what's going on.

Try, then, to name variables according to what they'll do:

- `name` is good, unless you need to separate out `first_name` and `last_name`.
- `id` is a bit short. So I often use `id_number`, or even `employee_id` or `national_id`.
- Most people would just use `email`, but I'm old fashioned and long winded, so I often write `email_address`.

As you can see from the preceding examples, there are a few conventions for Python variable names:

- With rare exceptions, only use lowercase letters.
- If you want to have multiple words, use _ between them.

There are some actual rules for variable names, though:

- You can use nearly any combination of letters, digits, and underscores (_).
- Capital letters are different from lowercase letters, so `x` and `X` are totally different variable names — but really, the convention is to always use lowercase.
- You cannot start a variable name with a digit.
- You can have _ at the start or end of a variable name, but you shouldn't do that, because those names are seen by Python (and

other developers) as special.

- Out of the box, Python defines *built-in* names, including `print`, `sum`, and `id`. You can define variables with these names, but it's a bad idea.
- Python also has a number of *keywords*, such as `for` and `while`, that you cannot use as variables. You'll get an error if you try.

And yes, there have definitely been times when I've been so frustrated by a software project that I've started to name variables things like `stupid` and `idiot`. It feels great at the time, but you're unlikely to rename them, which means that sometime down the road you'll be forced to understand what such variables really do.

Python Variables Are Different

Some introductory programming texts describe variables as folders, or boxes, with labels on them. When you assign `5` to `x`, they say, you're putting `5` in the box labeled `x`. And when you retrieve from `x`, you're retrieving the value from that box.

Some languages, such as C, actually work that way. In such languages, a variable is just an alias for a location in memory. Saying `x = 5` means that you are sticking the value `5` into a particular place in the computer's memory. Just like you can stick the value in the box.

Python's variables don't work this way at all.

For one, Python's values are never stored *in* a variable. Rather, the variable refers to the value. Any number of variables can refer to the same value. That's why it's useful to think of them (as you read earlier) as pronouns. More than one pronoun can refer to the same person, place, or thing. In the same way, any number of variables can refer to the same value.

But more than that: because values aren't stored in a variable, the variable doesn't need to know what kind of value it's storing. In many languages,

you need to "declare" a variable with a particular type, to ensure that the language will allocate the right amount of memory for the values it'll hold. An integer value needs more memory than a string value, for example. But in Python, all variables are the same size, because they just need to refer to the value's location elsewhere in memory. This is also why any Python variable can refer to any value, of any type — something known as *dynamic typing*.

One of my favorite websites, and one that I use a lot in this book to illustrate points, is Python Tutor, which visualizes Python code. I put the code

```
x = 5
y = 5
```

into the Python Tutor, and then ran it. By default, it displayed things in much the way that many people assume variables are stored, as shown in Figure 2-1.

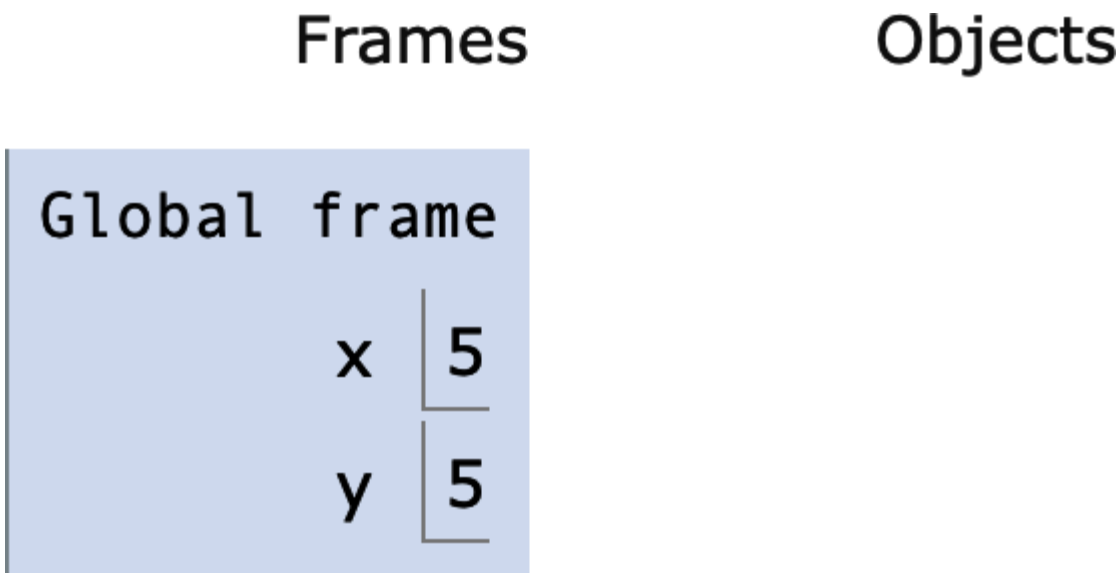


Figure 2-1. The wrong way to think about variable assignment

NOTE

The Python Tutor shows all of the values, or objects, on the right side of the screen, under the "Objects" heading. Variables, which refer to these objects, are off to the left, organized into *frames*, or collections of variables. The "Global frame" describes variables defined outside of a function; you'll learn more about functions and their variables in Chapter 9.

I understand why the Python Tutor does this, because when you have many variables and values, things can get very messy. But it misrepresents the way that things really work in Python. To see things the real way, select the "show strings and numbers as objects" checkbox, shown in Figure 2-2.

Write code in Python 3.11

```
1 x = 5
2 y = 5
3
```

Visualize Execution

Are you 60 or older? Help our research on older adults learning to code

teachers get free access to ad-free/AI-free mode

Python options (click for help):

- [ ] show strings and numbers as objects
- [ ] show list-of-lists as 2D array

show advanced options

Figure 2-2. Ensure that Python Tutor displays things in the right way

With this checkbox selected, the visualization changes, as you can see in Figure 2-3.

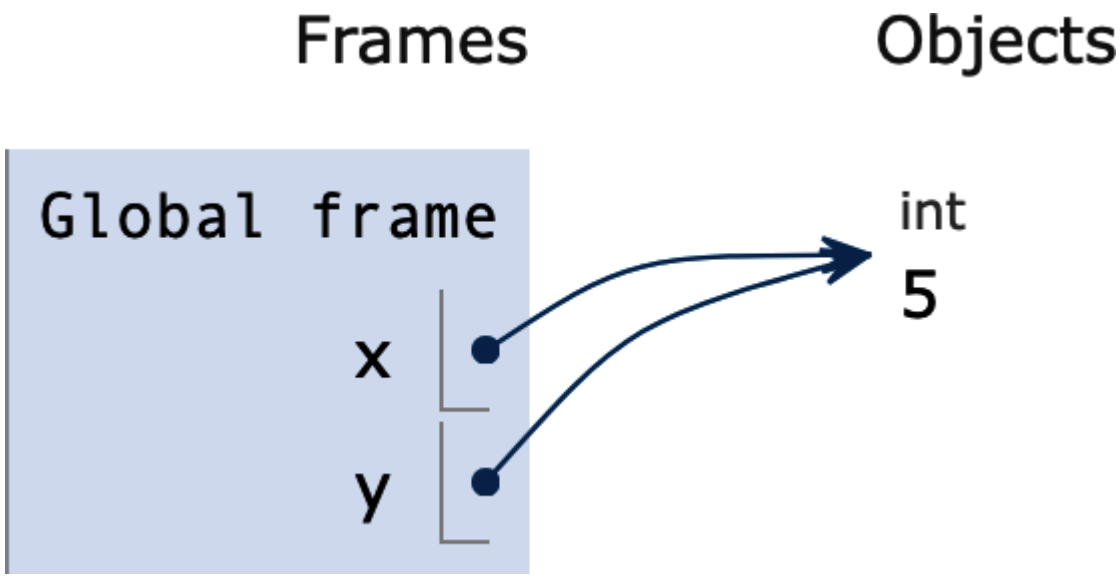


Figure 2-3. How to think about variables and values

Notice a few things about this diagram:

- Variables are on the left, and values are on the right.
- Without a variable, a value doesn't actually have any name.
- More than one variable can refer to the same value.

As you work with more complex code, values, and combinations of variables, this model — variables referring to values, not to other variables — will continue to be an accurate way to think about variables and values.

Exercise: Greeting / Strategy

It's time for you to practice what we've learned so far! Every exercise in this book has three parts:

- A programming problem
- An AI challenge

- My solution

The problem and solution are fairly obvious and standard — but what do I mean by an *AI challenge*?

It's an activity that I want you to do with a generative AI (GenAI) system. It doesn't really matter which GenAI chatbot you use; I regularly work with ChatGPT, Claude, and Gemini, and I believe that any of them will do the job just fine.

> **TIP**
>
> The job of the chatbot in this book is *not* to write the code for you. Rather, it's to accelerate and deepen your learning experience by giving you interactive feedback on your ideas, plans, coding, and understanding. I have about 10 different types of AI challenges that I'll use in this book, and my hope is that you'll not only use them to learn more from each exercise, but that you'll use them as models for how you can use GenAI after you read this book to learn and explore new topics.

Exercise: Greeting

For this exercise, assign your name to the `name` variable and the city in which you reside to the `city` variable. Using these two variables, display a simple greeting on the computer screen.

AI Challenge: Strategy Session

For many people, the hardest thing about programming is knowing where to start — how to break the problem down, how to decide which data structures and functions to use, and how to put it all together. For this reason, it's often good to have a plan before you start coding, and writing that plan down can force you to consider where it might have problems.

For this exercise, I want you to tell the GenAI chatbot what you plan to do in order to solve the problem. Be as specific as you can, but also tell the chatbot in your prompt that you want it to react and respond to your strategy at a high level, without writing any code. (If you don't say this, then it might write code, just to be helpful and friendly.)

I would also recommend telling the chatbot that you're only in Chapter 2 of this book, which will ensure that it doesn't use techniques that you haven't yet learned.

It might take several iterations to agree with the chatbot regarding how you plan to write your program. Once you and the chatbot agree that the strategy is sound, go ahead and write the code.

You can solve this exercise interactively, with the help of an AI tutor.

AI Challenge: Prompt and Solution

First, here's the strategy prompt that I handed to ChatGPT:

```
I'm on chapter 2 of an introductory Python book. The exercise I'm
working on says, "For this exercise, assign your name to the
`name`
variable and the city in which you reside to the `city`
variable. Using these two variables, display a simple greeting on
the
computer screen." I am putting together a strategy to solve this
problem, and want your feedback--not code!--before starting to
write Python code.

My plan is as follows:

* Assign a string value, my name, to the `name` variable. Because
my
  name is a string, I'll put it in quotes. I'll use the `=`
operator
  to assign that value to the `name` variable.
* On a second line, assign a second string value, my city, to the
  `city` variable. Because my city is also a string, I'll put it
in
  quotes.
* I'll then invoke the `print` function to display text on the
  screen. I'll use several `+` operators to join a greeting, my
name,
  a description, and my city's name together into a new string,
which
  I then pass to `print`.
* The greeting should then be displayed on the screen.

Am I missing anything from my strategy here?
```

ChatGPT:

```
Your strategy is clear and solid — you've identified the key
steps:
creating the two variables, using strings and the = assignment
operator, and displaying the output with print.

That said, I see a couple of small points you might want to
include or
think about:

Spacing and punctuation in concatenation – If you use + to join
strings, you'll need to make sure you add spaces (for example,
"Hello " + name rather than "Hello"+name), or else your output
might run words together.

Readability alternatives – You're correct that + works, but you
could also consider string formatting (like f-strings or .format)
for cleaner, more readable code. Even if your book hasn't
introduced these yet, it's good to be aware they exist.

Purpose of variables – Make sure your name and city variables are
descriptive enough and that the greeting clearly combines them
into a single, meaningful message.

Execution order – Variables must be assigned before they're used
in print. You've got this in the right order already, but it's
worth explicitly noting it as part of your plan.

If you like, I can also suggest how to tighten your plan into a
step-by-step checklist so you can apply the same thinking to
future
exercises. Would you like me to do that?
```

Notice that the chatbot is already hinting at other, better ways to accomplish the same task. (It correctly suggests using f-strings, a topic you'll cover in Chapter 3, which make such combinations of static and dynamic text easier to write and read.)

Also notice that even for a trivially easy program, there are many decisions to make, things to consider, and things that might go wrong.

As for the code itself, this is what I came up with:

```
name = 'Reuven'
city = 'Modiin'

print('Hello, ' + name + ' from ' + city + '!')
```

If you think that using + so many times inside your `print` call makes things a bit ugly, you're not alone. In the next chapter, you'll discuss f-strings (*format strings*), which ChatGPT alluded to.

You can solve this exercise interactively, with the help of an AI tutor.

Exercise: Calculator / Interview

Here, you'll use your expensive computer to do the job of a cheap calculator.

Specifications

Assign one integer (whole number) to the variable `first` and a second one to the variable `second`. Add these together, and put the result in the variable `total`. Print the value of `total`.

AI Challenge: Interview

This is one of my favorite GenAI techniques, and it can be used with any sort of Python code, no matter the level. For this to work, you need to already have working code. Feed it into the chatbot, and then ask it to quiz you on the program, to help reinforce your understanding of Python. If you answer incorrectly, it should correct you. If your answer is partly correct, then it can help to fix what you got wrong.

I would still recommend telling the chatbot that you're in the first part of an intro Python book, so that it doesn't ask you questions that are too terribly difficult. The prompt should ask the chatbot to ask you questions repeatedly until you tell it to stop.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

I first wrote my code:

```
first = 123
second = 456

total = first + second

print(total)
```

Then I went to Gemini, and entered the following prompt:

Prompt:

```
I'm on chapter 2 of an introductory Python book. The exercise I'm
working on says, "Assign one integer (whole number) to the
variable
`first` and a second one to the variable `second`. Add these
together,
and put the result in the variable `total`. Print the value of
`total`."

I'm enclosing my solution here:

first = 123
second = 456
total = first + second
print(total)

In order to improve my understanding of Python, repeatedly ask me
questions about the program. I'll answer as best as possible, and
you should correct my answer as necessary.
```

Gemini:

```
That's an excellent way to learn! Your code correctly solves the
exercise. Let's start with a question to deepen your
understanding.

What is a variable in Python, and what purpose does it serve in
the
program you wrote?
```

Prompt:

A variable is a name we give to a value.

Gemini:

That's a great, concise way to put it! You've captured the core concept perfectly.

To expand on that a little, think of a variable as a label or a tag
that you attach to a piece of data. In your code:

- first is the label for the number 123.
- second is the label for the number 456.
- total is the label for the result of 123 + 456.

The purpose of a variable is to store and reference data. It allows us
to give a meaningful name to a value so we can use it later in our
program without having to remember the value itself. This makes our
code much easier to read, understand, and change. For example, if you
wanted to change your program to add 7 and 8, you only have to change
the first two lines, and the rest of the code will work just the same.

Ready for the next question?

In the line total = first + second, what does the = symbol do? It's
not the same as the equals sign in mathematics.

Prompt:

The = symbol indicates that the two sides are equal, just like in mathematics.

Gemini:

```
That's a very common initial thought, but in Python and most
other
programming languages, the = symbol has a different meaning than
in
math.

In Python, the = is the assignment operator. Its job is to take
the
value on the right side and assign it to the variable on the left
side.

Let's break down the line total = first + second:

- Python first calculates the value of the expression on the
right:
  first + second. Since first is 123 and second is 456, the
result is
  579.
- Then, it takes that resulting value, 579, and assigns it to the
  variable total.

So, it's not a statement of equality; it's an action telling the
computer to store a value. You can think of it as "total gets the
value of first plus second."

The next question is about the data itself.

The numbers 123 and 456 are examples of a specific data type.
What is
the name of this data type, and what makes it different from
other
types of numbers, like 3.14?
```

This kind of interview is one of the most powerful techniques for ensuring that you understand a subject. It doesn't matter how complex the subject is; the GenAI system will likely be able to ask you questions and judge your answers, explaining things more than well enough. True, AI isn't perfect, and you should take its responses with a grain of salt. But the fact that it'll always come up with new, different questions also makes it more effective than reviewing the program on your own.

Summary

In this chapter, you learned about, and started to work with, values — the nouns of the programming world. You specifically looked at two types of values, integers (whole numbers) and strings (text). You also learned how to assign a value to a variable, what variables are, and how important it is to give them good names. You also started to use AI to reinforce what you have learned. In the next few chapters, you'll dive into Python's values in greater depth, and will use AI in additional ways to help you learn even better.

Chapter 3. Taking Input and Making Decisions

In the last chapter, you saw that you can assign values to variables, perform some basic operations, and even display some results on the computer's screen.

But let's face it: those activities were pretty boring. That's in no small part because the outcome of the program was known from the outset. You did not get any input from outside the program. And each of your programs executed, in full.

In the real world, software can handle many different inputs from a variety of different sources — the user, files, databases, and networks, to name a few examples. And software is powerful because it can make decisions based on these inputs, choosing what it calculates, prints, or transmits to other programs.

Consider what happens when you log in to your bank's website: if you enter a correct username and password, then you can see information about your account. You cannot see information about other people's accounts, even if they use the same bank. And if you enter an incorrect username or password, then you can't even see your own information.

This chapter explores both of these fundamental parts of programming. I'll show you how you can get input from the user, how you can compare values, and how you can use those comparisons to make decisions — and thus change what your software calculates and does. While your programs will still be relatively short and simple, you will have added an important set of tools and concepts to your programming toolbox.

input() Function

As you saw in the last chapter, you can assign a name (a string) to the variable `first_name` and then use it to greet the user:

```
first_name = 'Reuven'
print('Hello, ' + first_name + '.')
```

But of course, not every user is named "Reuven". To greet the user with their own name, you'll need to ask them a question. Whatever answer they provide should then be assigned to the variable `first_name`.

In Python, you do this using the `input` function. When you call `input`, you put one string argument — the *prompt* shown to the user — inside the parentheses:

```
input('What is your name? ')
```

Notice that I put a space after the question mark. This ensures that when the prompt is displayed on the screen, there will be a space just before the user's answer.

The preceding code works, but it is not useful. When Python sees the `input` function, it'll stop, ask the question, wait for the user to answer… and then throw out whatever the user wrote.

For this reason, you normally want to put `input` on the right side of assignment, in order to capture the user's text in a variable. For example:

```
first_name = input('What is your name? ')
```

Remember that when you assign in Python, the right side executes before the left side. So when Python encounters the preceding line, it first executes `input`, pauses the program, and waits for the user to respond to the prompt. Whatever the user types is then turned into a text string and assigned to the `first_name` variable on the left.

Once that has happened, `first_name` refers to a value that is no more and no less a string than anything else you have assigned. The difference is

that the user has influenced the value of this variable. You can thus say:

```
first_name = input('What is your name? ')
print('Hello, ' + first_name + '.')
```

The user's input, no matter how long or how short, is returned from `input` as a single string. The response to `input` is concluded by pressing the `ENTER` key, which means that the input cannot contain more than one line of text. (That line could be very long, though!)

If the user presses `ENTER` immediately, without entering any text, then you get a text string, but one without any content. That is known as the *empty string*, and you write it as `''` (or `""`, if you prefer double quotes).

Type Conversion Issues

In the last chapter, you wrote a simple calculator program. How could you rewrite that calculator, adding numbers provided by the user?

```
x = input('Enter first number: ')
y = input('Enter second number: ')
total = x + y
print(total)
```

What happens if you run the preceding program?

```
Enter first number: 10
Enter second number: 3
```

The output:

```
103
```

What is going wrong? From Python's perspective, it is actually all just fine: the user entered two strings, the first one of which was assigned to `x` and the second one assigned to `y`. `total` was assigned the result of adding two strings together.

That makes some logical sense, but what if you really want to add numbers together? What can you do then?

Integers in Python have a type of `int`. If you have a string `s` containing digits, then you can call `int(s)`. The result is an integer based on the digits in `s`. So if `s` is the string `'1234'`, then `int(s)` will return the integer `1234`.

Here's a slightly different version of the preceding program, which gets integers from `x` and `y` before adding them together and assigning the result to `total`:

```
x = input('Enter first number: ')
y = input('Enter second number: ')
total = int(x) + int(y)
print(total)
```

The output:

```
13
```

It worked! Python printed `13` on the screen, rather than `103`.

However, this program has a significant hole in it: if the user's text contains non-digits, then it'll fail with an error. You will learn to check for errors later in this book — but for now, we'll assume that your users will all read the documentation and enter valid input.

> **NOTE**
>
> For now, remember that `input` always returns a string, and that if you want to treat the user's input as an integer, you must use `int` to convert it explicitly.

f-strings

The preceding program printed `total`. What if you want to print not just `total`, but the entire expression? You could try:

```
x = input('Enter first number: ')
y = input('Enter second number: ')
total = int(x) + int(y)
print(x + ' + ' + y + ' = ' + total)
```

But this fails, with the following error message:

```
TypeError: can only concatenate str (not "int") to str
```

In other words: Python knows how to combine two integers with +. And it knows how to combine two strings with +. But it cannot handle combining an integer and a string with +. (You saw an example of this in Chapter 2.)

This is a common problem, and there are ways to fix it. For example, just as you can use `int` to get an integer from a string, you can call `str` to get a string from an integer:

```
x = input('Enter first number: ')
y = input('Enter second number: ')
total = int(x) + int(y)
print(x + ' + ' + y + ' = ' + str(total))
```

But the final line of this program is ugly and hard to read. Besides, must you always call `str` when you want to combine non-string values with strings?

The solution is the *f-string*, short for "format string." (Although I like to call them *fancy strings*, myself.) An f-string is a string in every way, and can be used wherever regular strings are. The difference is that an f-string may contain any number of curly braces (i.e., `{ }`). And inside those curly braces, you can put a Python expression — a variable name, an operation, or even a function call. The expression will be evaluated, turned into a string, and included in the output string.

This means that no matter what values you have, including integers, you can put them into an f-string and expect them to be displayed appropriately. In other words, f-strings allow you to mix text and numbers in your output — something you haven't been able to do easily until now.

To indicate that you have an f-string, rather than a regular string, put `f` before the string's opening quote. For example:

```
x = input('Enter first number: ')
y = input('Enter second number: ')
total = int(x) + int(y)
print(f'{x} + {y} = {total}')
```

Because the curly braces can contain any Python expression, you can even dump the `total` variable altogether:

```
x = input('Enter first number: ')
y = input('Enter second number: ')
print(f'{x} + {y} = {int(x) + int(y)}')
```

I'll use f-strings in most of my code from now on, and I encourage you to use them as well. Beyond making it possible to include Python expressions in a string, you can also add *format codes* that influence how things are displayed. A list of many f-string capabilities can be found at *https://fstring.help*.

Comparison Operators

We've already encountered the + operator, which returns a new integer (when adding integers) or a new string (when adding strings). Python has many other operators, and among the most important are *comparison* operators. As you might have guessed from the name, comparison operators compare two values. The result of a comparison is always `True` or `False`.

NOTE

`True` and `False` are known as *boolean values* in programming, named for George Boole, a 19th-century mathematician. Use them in Python whenever you need a yes or no answer. Other languages often use 1 and 0 for these purposes, but in Python, while you can use 1 and 0, it is better to use `True` and `False`. Note that both `True` and `False` start with capital letters.

The most commonly used comparison operator is == (i.e., a double equals sign), which asks whether two values are equal. Notice that this operator is ==, and asks whether the left side is equal to the right side. By contrast, =, the assignment operator (a single equals sign), performs an action, assigning to a variable. For example, `10 == 10` returns `True`, while `10 == 5` returns `False`. And if the variable `x` has been assigned the integer `5`, then asking whether `10 == x` will also return `False`.

You can compare any two values with ==, but you'll only get `True` if they're actually equal, which almost always means they must be of the same type. For example, `'10' == 10` will return `False`, because the integer `10` and the string `'10'` are not the same.

If you want to know if two things are unequal (i.e., the opposite of ==), you can use `!=`, the *inequality* operator. Wherever == returns `True`, `!=` returns `False`, and vice versa. So `10 != 10` returns `False`, and `10 != 5` returns `True`.

Python also supports four other comparison operators. These indicate whether one value is greater or less than the other:

- <: *Less than*
- <=: *Less than or equal*
- >: *Greater than*
- >=: *Greater than or equal*

For the most part, you use these operators as you might expect from math, with `5 < 10`, `20 > 10`, and `10 <= 10` all returning `True`, and `5 <`

`5`, `100 > 200`, and `50 <= 30` all returning `False`.

Just as + can be used with both numbers and strings, so can comparison operators. It might be obvious what == and `!=` mean with strings, but what does it mean for one string to be "less than" or "greater than" another? Python compares them *lexicographically*, which is a fancy term that basically means, "alphabetically, taking into account numbers, symbols, and other characters on your computer."

If you stick with words that only contain lowercase letters, then you can think of it as being just alphabetical; things get stickier and less obvious when you include capital letters, symbols, and digits (all of which are seen as coming before lowercase letters), letters from non-Latin alphabets, or emojis.

This means that `'cart' < 'horse'` is `True`, `'taxi' < 'taxicab'` is `True`, and `'egg' < 'chicken'` is `False`.

Python will let you compare any two values with == and `!=`. But if you try to compare an integer and a string with < or >, you'll get an error. That's because you can easily say that the string `'10'` and the integer `10` are not the same. But you can't really say that the string `'10'` is bigger than the integer `10`, or vice versa — and thus, Python won't let you do it.

Intro to Conditionals: if and else

You earlier saw how programs are more useful and interesting when they can include input from the user, and not just values that are written into the program itself. There's one more way in which programs can get more useful, namely making *conditional execution*. That is, instead of the computer always executing all of your code, you can tell it to execute parts of your program only under certain conditions.

You've already seen this in the software you use every day. Here are some simple examples:

- Some of the text on your computer is displayed normally, and other text is **in boldface**.
- The shape of your mouse cursor changes depending on whether it's on a text field, a menu, or a button.
- Your avatar in a video game moves to the left when you hit the left arrow, and the right when you hit the right arrow.

Thanks to `if`, you can make code *conditional*, only executing when a particular condition is `True`. Here's the basic idea:

- The `if` statement tells Python that the next section of code is conditional.
- To its right, you put an expression that evaluates to `True` or `False`.
- If the expression is `True`, then the `if` block runs. If not, then it doesn't run.

Here's an example of `if` in action:

```
first_name = input('Enter your name: ')

if first_name == 'Reuven':
    print('Hello, boss!')
    print('Nice to see you!')
```

The preceding code first asks the user to enter their name. However, it only prints something if the user is named "Reuven". That's because `if` looks to its right, sees the condition `first_name == 'Reuven'`, and evaluates that to either `True` or `False`. If it's `True`, then the `if` block runs. If not, then nothing at all is printed.

> **NOTE**
>
> Python is *case sensitive*, meaning that capital letters and lowercase letters aren't the same thing. This is true for variables, where `x` and `X` are seen as completely different. But it's also true for text. So if the user enters `'reuven'`, it won't match the string `'Reuven'`. The capital `R` might seem like a small difference to you, but from Python's perspective, it's completely different.

If you want to tell the computer to do something when the condition is `False`, you can use an `else` statement. `else` comes right after `if`, but doesn't have a condition. That's because `else` is the mirror image of `if`. Either the `if` condition is `True`, and its block runs, or it's `False`, and the `else` block runs:

```
first_name = input('Enter your name: ')

if first_name == 'Reuven':
    print('Hello, boss!')
    print('Nice to see you!')
else:
    print(f'Hi, {first_name}.')
```

One, and only one, of these blocks will run: either the `if` block will run, because the `if` condition is `True`, or the `else` block will run, because the `if` condition is `False`. One of these blocks is guaranteed to run.

If you don't have an `else` block, then the `if` block either runs or it doesn't.

What code can you put in an `if` or `else` block? Just about anything you want! For now, that means assignment, `print`, `input`, and additional `if` statements. But there are very few restrictions on what code can go inside an `if` block. This means that you can make all sorts of code execute conditionally.

Blocks and Indentation

Even if you are completely new to programming, I hope that the preceding code looks fairly straightforward to you, and that you basically understand

what is happening.

But if you have ever dabbled with another programming language before, then the preceding code might seem a bit empty. Where are the curly braces (`{` and `}`), or perhaps the words `begin` and `end`, indicating where a block starts and ends? How does Python know what lines of code should be executed if the `if` statement's condition is `True`, and which should be executed when it's `False`?

This raises one of the aspects of Python that longtime users see as an advantage, but which newcomers often see as odd and frustrating: blocks of code are marked with indentation. That's right, indentation is a fundamental part of Python's syntax. It is not optional or left up to your sense of aesthetics.

That is, at the end of both the `if` and `else` lines, you put a colon character (`:`). This tells Python: I'm about to give you a block of code, and this block is associated with the current line.

Following the colon, you then have one or more lines, all indented. In theory, you can indent using any combination of spaces and tabs, so long as you're consistent within a block. In practice, everyone uses four spaces for each level of indentation. Moreover, no one in the Python world thinks about it very much! That's because coding tools and editors designed to work with Python know how to work with indentation, and will basically indent the code for you.

Code that's indented after the `if` statement is only executed if the `if` is `True`. To move something out of the block, such that it's always executed, just delete the indentation with the backspace or delete key, making the code flush with the first character in the `if`.

If you forget to indent, or indent inconsistently, you'll get an `IndentationError` message from Python:

- At the end of an `if` or `else` line, or any other line that begins a block, you must end the line with a `:` character.

- The next line begins an indented block.
- A block must contain at least one line. But there is no maximum number of lines. All must be indented.
- A block may contain just about any Python code you want, including additional `if` statements. If you do that, then the inner `if`'s block will be indented further.

Why does Python use indentation, rather than setting off blocks with curly braces or keywords? In part, because programmers in other languages both indent their code and use such markers. Guido van Rossum, who invented Python, decided that if you're already indenting blocks, then why should you also use special characters? The result is that what you see with your eyes is what Python sees when it executes your code.

Consider this code:

```
first_name = input('Enter your name: ')

if first_name == 'Reuven':
    print('Hello, boss!')
    print('What a great day!')
```

The preceding code prints two lines to anyone identifying as "Reuven". If the user enters any other name, they are greeted with silence.

By contrast, consider this code:

```
first_name = input('Enter your name: ')

if first_name == 'Reuven':
    print('Hello, boss!')
print('What a great day!')
```

The preceding code says "Hello, boss!" to anyone identifying as "Reuven", because the first invocation of `print` is indented, and thus conditional on the comparison between `first_name` and `'Reuven'`.

But the second call to `print` is not indented. It will thus be executed under all circumstances, regardless of what name the user entered.

The only difference between these two blocks of code is whether the final line is indented. As you can see, the indentation makes a huge difference on the program's behavior.

Exercise: Which Word Comes First?

This exercise brings together everything you've learned in this chapter: getting input from the user, comparing values, and making decisions with `if`.

Specifications

Ask the user to enter two words, and assign each word to a separate variable. You can assume that the user enters two different words (i.e., not the same one), and that both words only contain lowercase letters. Tell the user which word comes first alphabetically.

For example:

```
Enter first word: marriage
Enter second word: love

love comes before marriage
```

You can solve this exercise interactively, with the help of an AI tutor.

AI Challenge: Strategy Session

Once again, I want you to use a strategy session with a GenAI system before you start to write your code. Tell the chatbot how you plan to write your code, and ask for feedback. Remember that this is a planning session, meaning that you don't have to write much code. And be sure to tell the chatbot that you don't want it to write the code for you, but rather just react to your plans.

AI Challenge: Prompt and Solution

Prompt:

```
I'm on chapter 3 of an introductory Python book. The exercise I'm
working on says, "Ask the user to enter two words, and assign
each
word to a separate variable. You can assume that the user enters
two
different words (i.e., not the same one), and that both words
only
contain lowercase letters. Tell the user which word comes first
alphabetically." I have just learned about the `if` and `elif`
statements.

My plan is as follows:

* Use the `input` function to ask the user to enter a word, and
assign
  it to `first`.

* Use the `input` function a second time, asking the user to
enter a
  second word, and assign it to `second`.

* Use `if` to compare `first` and `second` using the `<`
comparison
  operator.

* If this returns `True`, then in an indented block following the
  `if`, I'll indicate that `first` comes before `second`.

* Otherwise, I'll use `else` to indicate that `second` comes
before
  `first`.

* In both cases, I'll use an f-string to display the words
themselves.

Am I missing anything from my strategy here?
```

ChatGPT:

```
Your strategy is excellent — you've clearly broken the problem
into
steps, and nothing crucial is missing. 👍

A few points to reinforce what you're already planning:

Variable assignment: Using input() twice and saving into first
and
second is exactly right.

Comparison: The < operator works with strings, and it compares
them
alphabetically (technically, by their Unicode/ASCII order). Since
the
problem says the words will be lowercase, you don't need to worry
about uppercase vs lowercase ordering.

Branching: An if/else structure is all you need, since there are
only
two possible outcomes (and the exercise guarantees the words are
not
identical). You don't need elif here.

Output: Using f-strings to show the actual words will make the
result
clear to the user, which is exactly what the exercise is asking
for.

So in summary: your strategy is complete.  If you wanted to
stretch a
bit, you could also think about what would happen if the words
were
the same (the exercise rules that out, but in real programs that
might
matter). In that case, an extra elif first == second: branch
could
make sense.

Would you like me to show you what the program would look like in
code, based exactly on your plan, so you can check it against
your
strategy?
```

You can solve this exercise interactively, with the help of an AI tutor.

Having validated my strategy, I then went ahead and wrote the program:

```python
first = input('Enter first word: ')
second = input('Enter second word: ')

if first < second:
    print(f'{first} comes before {second}')
else:
    print(f'{second} comes before {first}')
```

COMPARING NUMBERS VERSUS STRINGS

What happens if two strings both contain digits, and you try to compare them with < or >? On the face of it, there wouldn't seem to be any difference:

```
print('100' < '200')
```

The preceding code prints `True` on the screen, as you might expect. But what about the following:

```
print('1000' < '200')
```

This *also* prints `True` on the screen. Why?

When Python compares two strings, it does so lexicographically:

- It compares the first characters of the two strings. If one comes earlier, then that string is considered lower.
- If the first characters are the same, then it goes onto each successive character until finding that one comes first.
- If the strings are the same, then both < and > will return `False` — but <= and >= will return `True`.
- If one string is shorter than the other, and is identical to the first characters of the second string, then it is considered to come first.

This is very different from comparing two integers!

If you were to compare the integers `1000` and `200`, then obviously `200` is lower. But if you compare the strings `'1000'` and `'200'`, then the `'1'` in `'1000'` comes before the `'2'` in `'200'` — and Python reports that `'1000'` is before `'200'`.

This is an example of how different values can look one way to people, but are seen completely differently by Python.

Additional Conditions with elif

Consider a program in which you ask the user how they will be traveling to an upcoming Python convention:

```
travel = input('How will you travel? ')

if travel == 'airplane':
    print('Have a nice flight!')
else:
    print('Enjoy your trip!')
```

This works, but it assumes that there are only two possibilities — airplane and everything else. What if you want to have separate messages for people flying, people driving, and then everyone else?

One possibility would be to stick another `if` condition inside the `else` block:

```
travel = input('How will you travel? ')

if travel == 'airplane':
    print('Have a nice flight!')
else:
    if travel == 'car':
        print('Enjoy your drive!')
    else:
        print('Enjoy your trip!')
```

With the preceding code in place, you can distinguish between people flying, driving, and doing everything else. This works because `else` means, "if the above condition wasn't `True`." It's a bit convoluted, though, both to write and to read.

What if you want to distinguish between airplanes, cars, trains, and everything else?

```
travel = input('How will you travel? ')

if travel == 'airplane':
    print('Have a nice flight!')
else:
    if travel == 'car':
        print('Enjoy your drive!')
    else:
        if travel == 'train':
            print('Trains are great; have a good trip!')
        else:
            print('Enjoy your trip!')
```

For most people, the code has now become too complex to read, let alone to maintain.

Fortunately, Python provides an alternative, the `elif` clause. It basically means, "If the above condition was `False`, then check this one instead. `elif` clauses come after the `if` clause and before the (optional) `else` clause. `elif` does require a condition, unlike `else`, which simply means, "If none of the above conditions matched."

Using `elif`, the preceding code becomes much easier to understand:

```
travel = input('How will you travel? ')

if travel == 'airplane':
    print('Have a nice flight!')
elif travel == 'car':
    print('Enjoy your drive!')
elif travel == 'train':
    print('Trains are great; have a good trip!')
else:
    print('Enjoy your trip!')
```

The conditions are checked, one by one, and in the order that they appear. One, and only one, of the expressions returns `True`. If there is no `else`

statement, and all of the `if` and `elif` conditions are `False`, then nothing will happen.

The order of the conditions can be important if they overlap. For example:

```
x = 50
if x > 20:
    print('Bigger than 20')
elif x > 40:
    print('Bigger than 40')
elif x > 60:
    print('Bigger than 60')
else:
    print('Very big!')
```

The preceding code prints, "Bigger than 20," because the first condition (`x > 20`) was `True`. Once that happened, the other `elif` conditions, as well as the `else`, were ignored.

Flipping the order of the conditions produces a different result:

```
x = 50
if x > 60:
    print('Bigger than 60')
elif x > 40:
    print('Bigger than 40')
elif x > 20:
    print('Bigger than 20')
else:
    print('Very big!')
```

When this second version of the code runs, it prints, "Bigger than 40," because the `if` condition was `False`, but the first `elif` condition was `True`. It never even got to the `x > 20` condition.

Exercise: Guess My Age

Here's another exercise that will bring together the different topics you've been learning. This time, we'll pretend that the user is someone other than yourself, and that they have to guess your age.

Specifications

Here are the parameters of the game:

- Assign your age to the variable `age`.
- Ask the user to guess how old you are.
- Print one of three answers: `too low`, `too high`, or `you got it!`
- You can assume that the user will enter an integer.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

To solve this, I first set a variable, `age`, to be my current age, 55. Note that this must be an integer, to ensure that the comparison works correctly.

Next, I used `input` to ask the user to enter a guess. The `input` function returns whatever the user types as a string, which in this case means that it is assigned to `guess`. To ensure that the program compares an integer with `age`, I then used `int` to get an integer based on `guess`, and assigned it back to `guess`.

I then used `if`, `elif`, and `else` to check whether the guess was right, too low, or too high. Notice that because there are only three possibilities when you compare two numbers — they are equal, the first is smaller, or the second is smaller — I was able to use `if`/`elif`/`else`, and know that I had covered all of the possibilities.

```
age = 55

guess = input('How old is Reuven? ')
guess = int(guess)

if guess == age:
    print('You got it!')
elif guess < age:
    print('Too low! (But thank you for the compliment)')
```

```
else:
    print('Too high! (Do I really look that old to you?)')
```

AI Challenge: Find the Bug

This AI challenge is meant to be done *after* you have already finished writing the program. Once you have finished writing the program, you ask an AI chatbot to introduce a bug, which you then have to find. Reading code is hard, and many bugs are subtle — so forcing yourself to read code carefully, looking for changes that might affect its output in a negative way, can be a great learning experience.

Prompt:

```
I'm a beginner Python coder, going through chapter 3 ("Taking
input
and making decisions") of an intro Python book. I just wrote the
following program to practice my skills:

age = 55

guess = input('How old is Reuven? ')
guess = int(guess)

if guess == age:
    print('You got it!')
elif guess < age:
    print('Too low! (But thank you for the compliment)')
else:
    print('Too high! (Do I really look that old to you?)')

Please introduce a bug into this code, and ask me to identify the
problem. (Keep in mind that I'm still new at coding!) Tell me
where my
analysis is right and wrong. Keep doing this until I tell you to
stop.
```

Claude:

```
age = 55

guess = input('How old is Reuven? ')
```

```
if guess == age:
    print('You got it!')
elif guess < age:
    print('Too low! (But thank you for the compliment)')
else:
    print('Too high! (Do I really look that old to you?)')
```

Try running this code and see what happens. What do you think the problem might be? Take your time to analyze it - remember to think
about what type of data you're working with and what the program expects.

Give it your best shot at identifying the bug!

Prompt:

It looks to me as though you aren't taking the user's input and converting it to an integer before comparing age and guess. The two
values won't be ==, so we'll get to the < comparison. And that will
fail, because you cannot use < on a string and an integer.

Claude:

Excellent analysis! You're absolutely right on both points:

✅ Correct: You removed the int(guess) line, so guess stays as a string
while age is an integer

✅ Correct: The == comparison between a string and integer will always be
False in Python

✅ Correct: When it gets to the elif guess < age: line, Python will throw a
TypeError because you can't compare a string to an integer with <

You've got a really good understanding of data types and how Python
handles comparisons! The error message would be something like
"TypeError: '<' not supported between instances of 'str' and

```
'int'".

Ready for another bug? Here's your code with a different problem:

age = 55

guess = input('How old is Reuven? ')
guess = int(guess)

if guess = age:
    print('You got it!')
elif guess < age:
    print('Too low! (But thank you for the compliment)')
else:
    print('Too high! (Do I really look that old to you?)')

What do you think is wrong this time?
```

Prompt:

```
The problem is that you're using =, the assignment operator,
rather
than ==, the equality comparison operator.
```

Claude:

```
Perfect! You nailed it again!

✅ Exactly right: I changed == (equality comparison) to =
(assignment operator)
in the if statement. This would cause a SyntaxError because
Python expects a
condition that evaluates to True or False in an if statement, not
an assignment.

You're really getting the hang of spotting these bugs! You
clearly
understand the difference between assignment (=) and comparison
(==),
which is a common source of confusion for beginners.
```

Combining Conditions with and, or, and not

You now know how to ensure that code only executes if a particular condition is `True`. But life is often more complex than this, and sometimes you need the computer to take multiple considerations into account. Consider these examples:

- Show a list of files, but only if the user is both logged in and has administrator access.
- Offer the option of a discounted annual subscription if a new customer has chosen your (more expensive) monthly billing option.
- Play a notification sound if the user has received a WhatsApp message or if a calendar event is about to start.
- Offer a discount on your online courses if the user's email address ends in *.edu* or their computer is located in a developing country.

The first two are examples of "and" conditions, in which two things need to be true in order to execute code. You could write this code using an `if` statement for the second, and then a second `if` statement inside the first block:

```
if first_condition:
    if second_condition:
        print('Both are true!')
```

But this can get a bit unwieldy. Even worse, the second two bullet points describe "or" conditions, where the code needs to execute if one or both of them is true. You have not yet learned how to do that, or at least not clearly and cleanly.

You'll need to learn two additional keywords, `and` and `or`:

- `and` takes one condition on its left and another on its right. If they're both `True`, then it returns `True`.

- `or` similarly takes two conditions, one on its left and another on its right. It returns `True` if one or both is `True`.

Just as `True` and `False` are known as *boolean values*, the operators `and` and `or` are known as *boolean operators*, because they work with `True` and `False` values.

Here's some example code:

```
x = 10
y = 20

if x == 10 and y == 20:
    print('Yes! Both are what you want!')

if x == 10 and y == 30:
    print('Yes! Both are what you want!')
else:
    print('No, one or both are not what you want')
```

The preceding code will print the "Yes" sentence for the first test, and the "No" sentence for the second.

We can similarly use `or`:

```
x = 10
y = 20

if x == 10 or y == 20:
    print('Yes! One or both are what you want!')

if x == 10 or y == 30:
    print('Yes! One or both are what you want!')

if x == 200 or y == 300:
    print('Yes! At least one is what you want!')
else:
    print('No, neither is what you want')
```

The preceding code will print "Yes" for the first test (since both conditions are met), "Yes" for the second test (since `x` is `10`), and "No" for the third test (since neither condition is met).

COMBINE EXPRESSIONS, NOT OPTIONS

I often see people write code like this:

```
if x == 10 or 20:
    print("Yes, x is either 10 or 20")
```

Many people think that the preceding code is saying, "If `x` is either `10` or `20`, then print the sentence." But that's not what it's saying! Both `and` and `or` require that they sit between expressions with either `True` or `False` values. In the preceding code, you have the expression `x == 10` to the left of `or`, which can be either `True` or `False`, depending on the value of `x`. But to the right of `or` is `20`, which is considered "truthy" in Python.

When `if` looks to its right, it must see a `True` or `False` value. What happens if it sees a value of another type, such as `20` or `'hello'`? It applies Python's rules for "truthy" values — meaning, which values are treated as if they are `True` (even though they are not), and which are treated as if they are `False` (even though they are not).

The general rule is that everything is considered truthy except for a number of values, known as "falsy":

1. The special value `None`
2. `False`
3. 0
4. Any empty value — for example, the empty string, `''`

In the above (problematic) code, the `if` condition first checked if `x == 10`, which was `True`. But on the other side of the `or` was just the integer `20`. `20` is not on the list of falsy values, so it returned `True` in this context.

In other words, the preceding code is effectively rewritten to be:

```
if x == 10 or True:
    print("Yes, x is either 10 or 20")
```

And of course, if you have `True` on either side of an `or` operator, the whole thing is `True`. Which means that the preceding code will always print the sentence.

For the effect you want, you really need to use the comparison operator `==` on either side of the `or`:

```
if x == 10 or x == 20:
    print("Yes, x is either 10 or 20")
```

Another way to handle this would be the `in` operator on a list or tuple, which we'll discuss in Chapter 4.

A close cousin of `and` and `or` is `not`. The whole purpose of `not` is to flip the logic on whatever sits to its right. If `x` is `True`, then `not x` will be `False`, and vice versa.

You can use `not` before any boolean value or expression. But it's often unnecessary. For example, you could say `not x == 10`, but it's more idiomatic to say `x != 10`, using the inequality operator `!=`.

I find that it's easier to read, write, and understand positive conditions than negative ones. This is particularly true when there's an `else` involved. For that reason, I tend to use `not` sparingly in my programs. If there is a way to express the same idea in a positive way, then I will try to do that.

NOTE

If you're coming from another programming language, then you might expect to use symbols, rather than words, for `and`, `or`, and `not`. But one of Python's most important values is code readability, and it was decided from a very early stage that `and` is easier to understand than `&&`, `or` is easier to understand than `||`, and `not` is easier to understand than `!`. Python does support `&` and `|` for mathematical operations on binary numbers, but you are unlikely to use those on a regular basis, so it's not worth worrying about them. The inequality operator `!=` might lead you to believe that there is a separate `!` operator. But no, you must use the word `not`.

Exercise: Name and Company

In this exercise, you'll use a combination of conditional code and boolean operators (`and`, `or`, and `not`) to give an appropriate greeting to someone based on their name and where they work.

Specifications

Here's what I want you to solve:

- Define two variables, `my_name` and `my_company`, populating them with two text strings — your name, and where you work.
- Ask the user to enter their name and store it in a variable, `user_name`.
- Ask the user to enter the name of their company and store it in a variable, `user_company`.
- Based on the user's answers, print one of four greetings:
  - If the name and company are the same as yours, greet yourself ("You must be me!").
  - If the name is the same as yours, but the company is not, compliment the name but be snarky about the company.

– If the company is the same as yours, but the name is not, greet your colleague.

– If neither of the user's inputs matches yours, then be snarky toward both their name and company.

I don't endorse being snarky to people in real life just based on their name or company — but hey, it's worth having some fun with this exercise.

Solution

The first thing that I did was to define four variables. Two are set with my personal information, and two are based on input from the user:

```
my_name = 'Reuven'
my_company = 'LernerPython'

user_name = input('Enter your name: ')
user_company = input('Enter your company: ')
```

Next, I used `if` to make a decision. The start was fairly clear:

```
if user_name == my_name and user_company == my_company:
    print('Hey, you must be me!')
```

So far, so good. But then I had to check whether the name was the same and the company was different. It's tempting to write the following:

```
elif user_name == my_name and user_company != my_company:
    print('Great name, but terrible employer!')
```

There is not anything wrong with this code; it does the job and works just fine. But consider the logic: the `if` clause checked if both `user_name` and `user_company` matched the user's values. In the `elif` clause, you checked if `user_name == my_name`. If that's `True`, then you know for sure that `user_company != my_company`. After all, if they were equal, then the `if` condition would have been `True`, and you wouldn't have reached the `elif`.

You can thus simplify the code, removing the `and` and the second comparison:

```
elif user_name == my_name:
    print('Great name, but terrible employer!')
```

Is this a good idea? That's a matter of taste. I like it when code is shorter and cleaner. But I also like it when code is clearer and more explicit. And thus, I'll admit that I'm torn over which version of the code is truly better, although I lean slightly in favor of the shorter version.

Here's how the next two conditions play out:

```
elif user_company == my_company:
    print(f'Greetings, {user_name}, my colleague!')
else:
    print('Your name and company are both terrible!')
```

If you're greeting either colleagues or strangers in this way, then you might want to put your Python studies aside and pick up a book on interpersonal communication.

The full code thus looks like this:

```
my_name = 'Reuven'
my_company = 'LernerPython'

user_name = input('Enter your name: ')
user_company = input('Enter your company: ')

if user_name == my_name and user_company == my_company:
    print('Hey, you must be me!')
elif user_name == my_name:
    print('Great name, but terrible employer!')
elif user_company == my_company:
    print(f'Greetings, {user_name}, my colleague!')
else:
    print('Your name and company are both terrible!')
```

AI Challenge: Explanation

This time, the GenAI task is seemingly simple: give the GenAI a copy of your code, and then explain what it does. Then ask the GenAI to evaluate your explanation.

Why is this helpful to your learning?

First, there's an old saying that the best way to learn something is to teach it. As a professional instructor, I've certainly found this to be true; only because I have to explain things on a regular basis do I really feel like I know it. Having to explain what you've done to the GenAI forces you to crystallize your thinking, and describe what you've done in a precise way.

Second, while you're explaining things, you'll undoubtedly discover that you missed something. Perhaps you did not take certain inputs into account. Or you did not check data types. Or there's a bug in the code. Or it doesn't precisely match the specification. Regardless, it's only through walking someone else through your work that you'll often pick up on such things.

Third, as you explain and talk about your code more, you'll discover recurring patterns. As you use and discuss these patterns, they'll become more prominent in your mind, more easily accessible the next time you need to solve a problem.

Here's a short session I had with ChatGPT about this problem:

Prompt:

```
I'm a beginner Python coder, going through chapter 3 ("Taking
input
and making decisions") of an intro Python book. The problem
statement is as follows:

In this exercise, you'll use a combination of conditional code
and
boolean operators (`and`, `or`, and `not`) to give an appropriate
greeting to someone based on their name and where they work.

* Define two variables, `my_name` and `my_company`, populating
them
  with two text strings--your name, and where you work
* Ask the user to enter their name, and store it in a variable,
`first_name`
```

* Ask the user to enter the name of their company, and store it in a
  variable, `company`
* Based on the user's answers, print one of four greetings:
** If the name and company are the same as yours, greet yourself ("You
   must be me!")
** If the name is the same as yours, but the company is not, compliment
   the name but dismiss the company
** If the company is the same as yours, but the name is not, say hello
   to your colleague
** If neither of the user's inputs matches yours, then be snarky
   toward both their name and company

I wrote the following code:

```
my_name = 'Reuven'
my_company = 'LernerPython'

user_name = input('Enter your name: ')
user_company = input('Enter your company: ')

if name == my_name and user_company == my_company:
    print('Hey, you must be me!')
elif user_name == my_name:
    print('Great name, but terrible employer!')
elif user_company == my_company:
    print(f'Greetings, {user_name}, my colleague!')
else:
    print('Your name and company are both terrible!')
```

I've been asked to explain the code to you, to help ensure that I understand what I've done. Here I go:

1. I define two variables, my_name and my_company, for easier comparison. I could have hardcoded these, too.

2. I get inputs from the user. The `input` function always returns a
string, so these values are strings.

3. I then check whether `user_name == my_name`, using the `==` operator for
equality, as well as the `and` operator that only returns `True` if
both conditions are `True`.

4. If they're both `True`, then the `if` block fires, printing a nice
greeting to myself.

5. If they aren't both `True`, then you check the next possibility with
`elif`. Here, I could have checked `user_name == my_name and company !=
my_company`, but I decided just to check `user_name == my_name` because we
already know that if that's true, `company != my_company`.

6. Similarly, my second `elif` line only checks if `company == my_company`.

7. The fourth and final logical possibility is handled with an `else`,
which doesn't take any condition. In that case, I give a snarky
greeting to the person.

What do you think of my explanation?

ChatGPT:

Your explanation is very good — clear, structured, and it shows that
you understand why the code works, not just what it does. Let me give
you some feedback, point by point:

What you did well

✅ Step-by-step reasoning: You broke down the program into its main pieces:
defining variables, taking input, comparing, and branching with if/elif/else.
That's exactly how to explain code clearly.

✅ Understanding of operators: You showed that you understand how ==, and,
and != work, and why you only need certain conditions in the later branches.

✅ Good awareness of flow: You noticed that by the time Python gets to the

second elif, we already know that the first condition wasn't true. That's
a key insight about how if/elif chains work.

✅ Awareness of else: You correctly explained that else covers all remaining
possibilities, so you don't need to write out a condition.

A few ways you could make it even stronger

Be explicit about order of execution You hinted at this, but you could
emphasize: Python checks the conditions in order, top to bottom, and
executes the first matching block. After that, it skips the rest.

Explain why strings work this way You mentioned input returns strings,
which is great. You could add that string comparisons with == are
case-sensitive, so "Reuven" is not the same as "reuven". That might
matter if the user doesn't type exactly what you expect.

Note about readability You already thought about whether to include
company != my_company in the second branch. Explaining that shorter
conditions improve readability (since we already know the earlier test
failed) shows good coding style awareness.

Stretch idea (optional for your level) If you wanted to be fancy, you
could mention that instead of hardcoding my_name and my_company in the
program, you could eventually load them from a file or environment
variable. But that's beyond Chapter 3.

👉 Overall: Your explanation shows you really get the key ideas of if/elif/else
and boolean logic. If you gave me this explanation in class, I'd say you're
not just memorizing but actually thinking like a Python programmer.

Summary

In this chapter, you learned some of the most essential parts of programming:

- Getting input from the user
- Making decisions with `if`/`elif`/`else`
- Combining conditions with the boolean operators `and` and `or`

Almost every Python program, no matter how simple or complex, uses all of these. That's why it's so important that you understand how they work, and the syntax used when writing them. In the next chapter, we'll dive deeper into the data types we've been using so far, namely integers and strings.

Chapter 4. Working with Numbers and Strings

The previous two chapters introduced the fundamentals of Python: you got input from the user, assigned values to variables, made decisions, and displayed results on the screen. You saw that there are some fundamental differences between integers and strings, and how to convert data from one type to another.

In this chapter, you're going to get a more formal, in-depth understanding of these types of data, often known as *data structures*. You will see how to define them, what their limits are, and how to work with them. Along the way, you will learn how Python's approach to both numbers and strings is different from many other programming languages, and how this can make Python more consistent and easier to understand.

Toward the end of this chapter, I will introduce the idea of *methods*, special-purpose functions that are attached to data structures. Python actually has many more methods than standalone functions. As a result, this chapter will not only help you to understand strings better, but will also help you understand how to work with numerous Python data structures.

Types of Numbers

Python supports two main types of numbers. Whole numbers, as you've already seen, are known as integers, with the type `int`. Numbers with a decimal point and an optional fractional part are known as "floating-point," with the Python type `float`.

If Python sees digits without any quotes, parentheses, or brackets around them, then it considers them to be integers. For example:

```
x = 123
y = 456789
z = -12345
```

Notice that Python supports both positive and negative integers. If the integer starts with `-`, then it is considered negative.

You cannot use a comma to separate digits for easier reading. But you can use an underscore (_):

```
x = 123
y = 456_789
z = -12_345
```

> **NOTE**
>
> The _ is ignored by Python when it reads the number; it's there for the sake of humans writing and reading the code. However, you cannot put _ at the start or end of the number. Nor can you put two _ characters in a row.

If the digits contain a decimal point, then the number is seen as a `float`. For example:

```
x = 123.45
y = 456.789
z = -123.45
```

Without a decimal point, a number is an `int`. If a number lacks a fractional part, then you can either put `.0` (or even just `.`) at the end to force it to be a `float`:

```
x = 123.0
y = 456.
```

Python will display both of the preceding numbers with `.0`, though:

```
print(x)     # 123.0
print(y)     # 456.0
```

You have already seen that `int` will return a new integer value from a string. But `int` will also work on a float, returning the integer portion:

```
int(123.45)   # 123
int(456.99)   # 456
int(-3.9)     # -3
```

Notice that `int` does not round the number to the nearest integer. Rather, it removes the fractional part.

If you have an integer or string, and want to get a float based on it, just invoke `float` on the value:

```
float(1)      # 1.0
float('2.3') # 2.3
```

If you perform a math operation on both an integer and a float, you will get a float back:

```
print(1 + 2.3)    # 3.3
```

NOTE

Why do computer people use the term "float" for numbers with a decimal point? Because the decimal point can move (or "float") to different positions. Computers store these numbers a bit like scientific notation, keeping track of the significant digits separately from where the decimal point belongs. This allows floats to represent both tiny numbers like 0.0000001 and huge numbers like 123,000,000.0 efficiently.

Why Two Numeric Types?

Why does Python need both `int` and `float`? Can't we just have "numbers" and be done with it? And besides, why do we need our own numbers, when the computer's lowest levels are implemented using binary digits?

Let's start with the second question: in a high-level language like Python, we don't think about or refer directly to the underlying computer hardware, or even the bits (binary digits) that it uses. Rather, we work with data structures that more closely reflect how we humans think and work. So yes, at the end of the day, everything in a computer uses binary numbers. But that does not mean we want or need to work directly with such numbers — and in Python, we don't.

In some languages, the numbers do reflect the underlying binary implementation. Users of those languages talk about 32-bit integers and 64-bit integers, with the number of bits reflecting the maximum value that an integer can hold. In Python, by contrast, we don't have such limits; an integer can contain absolutely any value you want, positive or negative. The number of digits is limited by the amount of memory in your computer.

Given Python's flexibility, why do we still need two separate types of numbers? Why not just use floats everywhere?

Mostly because floats are tricky and complex to implement. Because of that complexity, floating-point math operations are significantly slower than their integer counterparts. Indeed, computer speed is often measured in "FLOPS," or "floating-point operations per second," with today's fastest computers measured in "TeraFLOPS," or trillions of floating-point operations per second.

If you're doing lots of floating-point math operations, it's even useful to have a special chip installed that handles these operations. The leading manufacturer of such chips is NVidia, and while they got their start making chips for video games (which require lots of floating-point math), their recent rise is due to AI (which also requires lots of floating-point math).

So yes, Python could have been implemented with a single numeric type, one that would effectively be `float`. But that would slow the language down, and would limit the range of numbers that we can represent.

Numeric Operations

You have already seen that Python supports addition with the + operator:

```
x = 10
y = 20
print(x + y)    # prints 30
```

Python supports a variety of other operators, too, listed in Table 4-1.

Table 4-1. Integer operations in Python

Operator	Name	Example	Result
`+`	Addition	`10 + 3`	`13`
`-`	Subtraction	`10 - 3`	`7`
`*`	Multiplication	`10 * 3`	`30`
`/`	Truediv	`10 / 3`	`3.3333333333333335`
`%`	Modulo (int division remainder)	`10 % 3`	`1`
`**`	Exponentiation	`10 ** 3`	`1000`

For many people, the biggest surprise is that we have two separate operators for division, `/` ("truediv") and `//` ("floordiv"). Traditionally, programming languages assumed that if you divided one integer by another, you wanted to get an integer back. Indeed, even old versions of Python used to work this way. This meant that `10 / 3` would return the integer `3`, as would `9 / 3` and `11 / 3`. After all, you were performing an operation on integers, so it made sense to return an integer. If you wanted to get a non-integer

result, the argument went, you would have converted one or both of those numbers into floating-point numbers.

For most humans, this seemed counterintuitive. And so, in modern Python the `/` operator always returns a float. If you divide `10 / 10`, you get `1.0` back. And if you divide `10 / 4`? You get `2.5`. That's what most people would expect, and it has thus become the default, standard behavior for `/`.

If you want to remain within the world of integers, you're welcome to use `//`, known as "floordiv." As its name implies, "floordiv" does not round the result to the closest integer. Rather, it removes the fractional part entirely, returning an integer.

Floats support all of the same operations as integers. However, if you use `//` with floats, you will get a float back, albeit one with `.0` as the fractional part.

It's common to use these operators on the right side of an assignment (=) operation. For example:

```
text = input('How old are you? ')
age = int(text)

print(f'This year, you are {age}.')
age = age + 1
print(f'Next year, you will be {age}.')
```

The first few lines of the preceding code aren't that surprising: it asks the user to enter their age with `input`, converts that value to an integer with `int`, assigns the resulting integer to `age`, and prints the current age.

But then, to find the user's age *next* year, I used `age = age + 1`. Remember that in assignment, the right side is evaluated, and its value is assigned to the variable on the left. That line of code should thus be read as, "add 1 to the current value of `age`, then assign the result back to `age`."

This means that when the final line of code is executed, `age` will have a new value, 1 more than whatever the user entered. It'll thus print an accurate value for the user's age next year.

However, there is a common shorthand way to write `age = age + 1`. You can instead write `age += 1`, which has the same effect, adding 1 to `age` and assigning the new value back to the variable `age`.

> **NOTE**
>
> Many other languages support another way to add 1 to a variable, with the `++` operator. In such languages, you can say `age++`, and 1 would be added to the variable `age`, same as `age += 1`. Python does not support `++` or subtraction `--`; if you want to use a special operator to increment a variable's value, you will need to use `+=`.

Exercise: Guessing Game

In this exercise, you will implement a very simple game, one that will reinforce some of the ideas you've been learning about working with numbers.

Specifications

Here's how the game should work:

- Define a variable, `number`, with the number that the user needs to guess. It should be an integer between 0 and 100.
- Ask the user to guess the number. Assign the guess to the variable `guess`.
- Give one of three responses:
  - You got it!
  - Too low
  - Too high

Note that the user will only have one chance to guess the number. After that one guess, the game is over.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

I first defined `number` to be 72, with the assignment operator, =:

```
number = 72
```

Next, I asked the user to enter their guess:

```
guess = input('Guess the number: ')
guess = int(guess)
```

Remember, though, that `input` always returns a string value. At this point, `guess` thus contains a string, even if the user only entered digits. To get an integer, I ran `int` and assigned the result back to `guess`:

```
guess = input('Guess the number: ')
guess = int(guess)
```

With `guess` assigned an integer, the program can finally decide what to show the user:

```
if guess == number:
    print(f'You got it!')
elif guess < number:
    print(f'Too low!')
else:
    print(f'Too high!')
```

Put it all together, and the program looks like this:

```
number = 72

guess = input('Guess the number: ')
guess = int(guess)

if guess == number:
    print(f'You got it!')
elif guess < number:
```

```
    print(f'Too low!')
else:
    print(f'Too high!')
```

Here's an example run of the program:

```
Guess the number: 72
You got it!
```

Here's another run:

```
Guess the number: 15
Too low!
```

What happens if you leave out the conversion line, in which Python turns `guess` from a string into an integer? Here's the output that I got from running it, where I guessed `72` — the number that is in `guess`:

```
Guess the number: 72
Traceback (most recent call last):
  File "", line 7, in
    elif guess < number:
         ^^^^^^^^^^^^^^
TypeError: '<' not supported between instances of 'str' and 'int'
```

When Python gives us an error message, it tries to give us some context, indicating where the error occurred and what the problem was. In this case, it points to line 7 of the program as the culprit. After showing us the context, emphasized with `^` characters under the offending code, it then writes `'<' not supported between instances of 'str' and 'int'`.

There are two problems with this error message: first, it's hard for newcomers to understand. Second, it does not really point to the problem.

Consider what happens in this version of the program, step by step:

- `number` is assigned the integer value `72`.
- The user's guess, the string `'72'`, is assigned to `guess`.

- Python compares `number` and `guess`. Are they the same?
- No, they aren't! While `72` and `'72'` seem identical to humans, they are of different types, and thus unequal.
- Since the `if` condition is `False`, Python moves on to the `elif` condition, comparing `guess < number`.
- Just as you cannot use + on an integer and a string, you cannot use < on an integer and a string. The program exits.

The error message wasn't wrong, per se. It's true that Python tried to compare `guess < number`, and because those two types aren't comparable, Python produced an error.

But the only reason that this comparison was made was that `guess` and `number` were of two different types, and thus == returned `False`. When the `if` condition is `False`, Python moves on to the next condition, which in this case was the use of < inside `elif`.

> **NOTE**
>
> This example shows not only how important it is to keep track of data types, but also that error messages can sometimes be misleading. When you get an error message, realize that the problem might stem from some code that executed earlier, setting up the problem that is being reported.

AI Challenge: Add a Bug

Do you feel comfortable with the code you wrote? This time, the AI challenge is to add a bug to it, and to ask an AI chatbot to figure out what the bug is.

The point of this exercise isn't to see if AI can find a bug. It usually can. Rather, it's to see if you understand the code well enough to know how to break it, and what the effects of your breakage will be. For this reason, you want the AI to not only find the bug, but tell you what harm it'll do. You

should then run the code and see if your prediction, or that of the AI, is correct.

Here is an example of an interaction I had with ChatGPT:

```
I'm in the 4th chapter of "Python for non-programmers," which
talks
about numbers and strings. I have just written a program as part
of an
exercise. However, I have introduced a bug into the code. Can you
find
it, and tell me what problems I will see as a result of the bug?

Number = 72

guess = input('Guess the number: ')
guess = int(guess)

if guess == number:
    print(f'You got it!')
elif guess < number:
    print(f'Too low!')
else:
    print(f'Too high!')
```

Here is what ChatGPT said to me:

```
🐛 The bug

You defined the variable as Number (with a capital N), but in
your if and
elif lines you refer to number (with a lowercase n).

Python is case-sensitive, so Number and number are treated as
different
variables.

💥 What will happen

When you run the code, Python will raise an error as soon as it
hits the line:
```

And indeed, that was the bug that I introduced. ChatGPT did a great job of finding and analyzing the bug.

What sorts of bugs can you think of? Does GenAI correctly find them and analyze them?

Strings

Let's now switch gears, and talk about how to work with text in Python. You've already seen that all text in Python is stored in "strings." You can define a string with either single quotes (`'`) or double quotes (`"`), so long as the quotes at the start and end of the string match. The shortest string, known as the "empty string," contains zero characters, and is written as `''` or `""`. There is no maximum string length, meaning that strings can be used for everything from a single character to an entire encyclopedia. Python, unlike some other programming languages, does not have a "character" data type.

There is no functional difference between single and double quotes in Python. You can thus say:

```
s1 = 'Hello out there!'
s2 = "Hi, again!"
```

If your string contains one or more quote characters, then it's often easiest to just use the other type of quote to define the string. For example:

```
s3 = 'He said, "Hello."'
s4 = "She's the new CEO."
```

If you prefer, you can put a backslash (`\`) before a quote character inside a string. That tells Python that you want the quote to be used as a regular character, rather than be seen as a delimiter marking the start or end of the string. This is known as *escaping* the character. For example:

```
s5 = "He said, \"Hello.\""
s6 = 'She\'s the new CEO.'
```

I find this to be a bit ugly, though, and avoid it unless I have no other choice. For example, if a string contains both single and double quotes, one of them will need to be escaped. For example:

```
s7 = 'He said, "She\'s the new CEO."'
```

Both single and double quotes can be used to create an f-string.

Thanks to a standard known as "Unicode," Python strings can handle just about any character you throw at it. This includes not only characters from languages as different as English, Arabic, and Chinese, but also musical notes and even emojis.

> **NOTE**
>
> What does the word "string" have to do with text? In a computer's memory, each character was traditionally stored in a single byte in the computer's memory. Combining several of these characters together made it possible to store a word, sentence, or longer text. These combinations were sometimes described as being "strung together," much like pearls on a necklace. The term "string" stuck, and is pretty universal among programmers.

Indexes

To retrieve a single character from a string, use square brackets (`[]`). Inside the brackets, put the index of the character you want to retrieve, such as `[3]`.

There is an important catch, however: Python, like many programming languages, starts counting with 0. That is, the first character will be at index 0, the second character at index 1, and the third character at index 2. To print the first three characters of a string `text`, you can thus say:

```
text = 'Hello'
print(text[0])
print(text[1])
print(text[2])
```

The result:

```
H
e
l
```

Each character in a string has an index, starting with 0, as you can see in Figure 4-1.

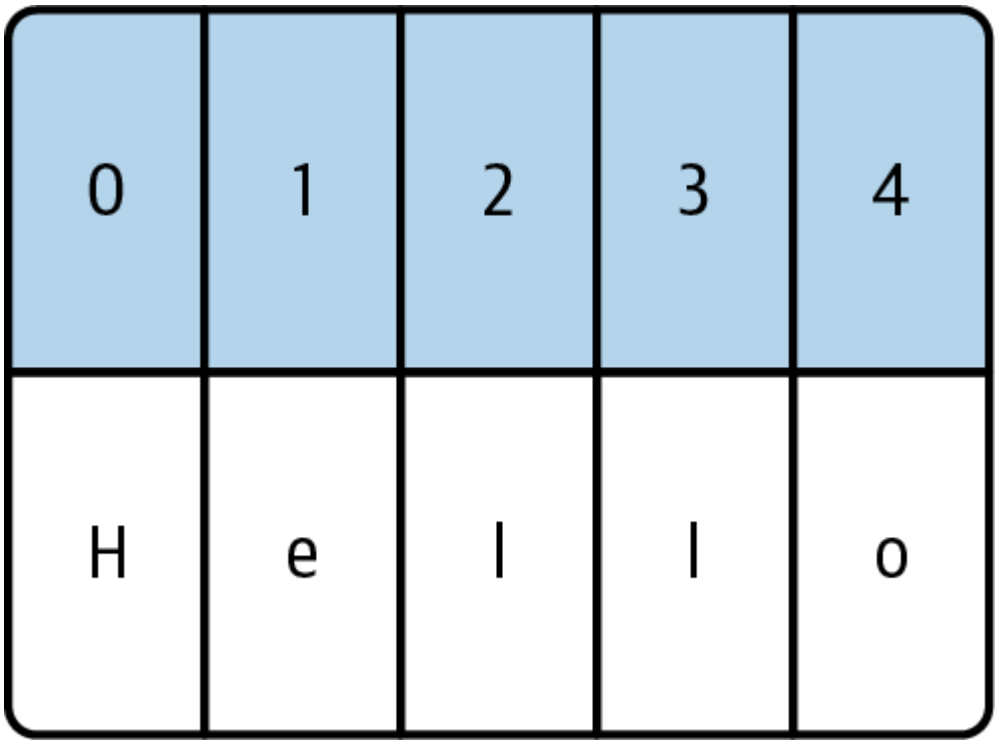


Figure 4-1. How to think about string indexes

If you ask Python to retrieve a character from an index that does not exist, you will get an `IndexError` exception:

```
text = 'Hello'
print(text[100])
```

The result:

```
Traceback (most recent call last):
  File "", line 2, in
    print(text[100])
```

```
          ~^^^^^^
IndexError: string index out of range
```

The error message points (with ^ characters) at the place that triggered the error, while telling you, "string index out of range."

You can use the built-in `len` function to calculate a string's length. It returns an integer, the number of characters:

```
text = 'Hello'
length = len(text)
print(f'The string "{text}" contains {length} characters.')
```

This prints:

```
The string "Hello" contains 5 characters.
```

Notice how the preceding code uses an f-string. The f-string contains two dynamic portions inside curly braces (`{}`), both of which display variable values. It's possible to shorten the preceding code:

```
text = 'Hello'
print(f'The string "{text}" contains {len(text)} characters.')
```

This is possible because the `{}` in an f-string can contain any Python expression, including the result of invoking a function.

The integer placed in `[]` can be stored in a variable, or the result of a calculation. For example:

```
text = 'Hello'
i = 2
print(f'The item at index {i} in {text} is {text[i]}.')
```

The preceding code prints:

```
The item at index 2 in Hello is l.
```

`len` returns the number of characters, but the index starts at 0. This means that the final character will always have an index one less than the string's length. For example:

```
text = 'Hello'
print(f'The final character in "{text}" is {text[len(text)-1]}.')
```

The preceding code prints:

```
The final character in "Hello" is o.
```

If you find yourself a bit confused by Python's zero-based indexing, you aren't alone. One of the most common mistakes made in programming is called an "off-by-one error," and it most commonly happens when we fail to add (or subtract) one from the index.

It's also common to forget that every character counts, even the ones that you don't see. For example, what is the length of the string `'abc def'`? The answer is 7:

```
text = 'abc def'
print(f'The length of "{text}" is {len(text)}.')
```

The result:

```
The length of "abc def" is 7.
```

From Python's perspective, the space character in the middle of the string is no more and no less important than any of the others. There is a category of characters known as "whitespace" that aren't displayed on the screen, but which do consume memory and a position in a string. Consider this code:

```
s1 = 'abc'
s2 = 'abc '

if s1 == s2:
    print('They are the same')
```

```
else:
    print('They are not the same')
```

If you run the preceding code, it will print, "They are not the same." That's because `s1` contains the three characters `'abc'`, whereas `s2` contains the four characters `'abc '`. From Python's perspective, these are two completely different strings.

You already saw that asking for an index higher than the `len(s)-1` will generate an error. But what about negative indexes? That is, how will Python respond if you ask for `s[-1]`?

```
text = 'Hello'
print(f'The final character in "{text}" is {text[-1]}.')
```

This prints:

```
The final character in "Hello" is o.
```

As you can see, `text[-1]` returns the final character of `text`. Whereas indexes normally start with 0, get up to the length – 1, and retrieve from the left side of the string, *negative* indexes start with -1, get up to the length of `-len(s)`, and retrieve from the right of the string. For example:

```
text = 'Hello'
print(f'text[-3] in "{text}" is {text[-3]}.')
```

The result:

```
text[-3] in "Hello" is l.
```

Slices

What if you want to retrieve more than one character from a string? For example, what if you want to retrieve the first three characters? Or the third

and fourth characters? In such cases, you can use a "slice," which looks like this:

```
text = 'abcdefghijklmnopqrstuvwxyz'
print(text[10:20])
```

The code prints:

```
klmnopqrst
```

To get a slice from a string, use `[]`, just as if you were retrieving a single index. However, you will actually use *two* numbers, separated by a colon (`:`). Before the colon, put the index from which the slice should start. After the colon, put the index at which the slice should end. You can see how Python thinks about the slice text[10:20] in Figure 4-2.

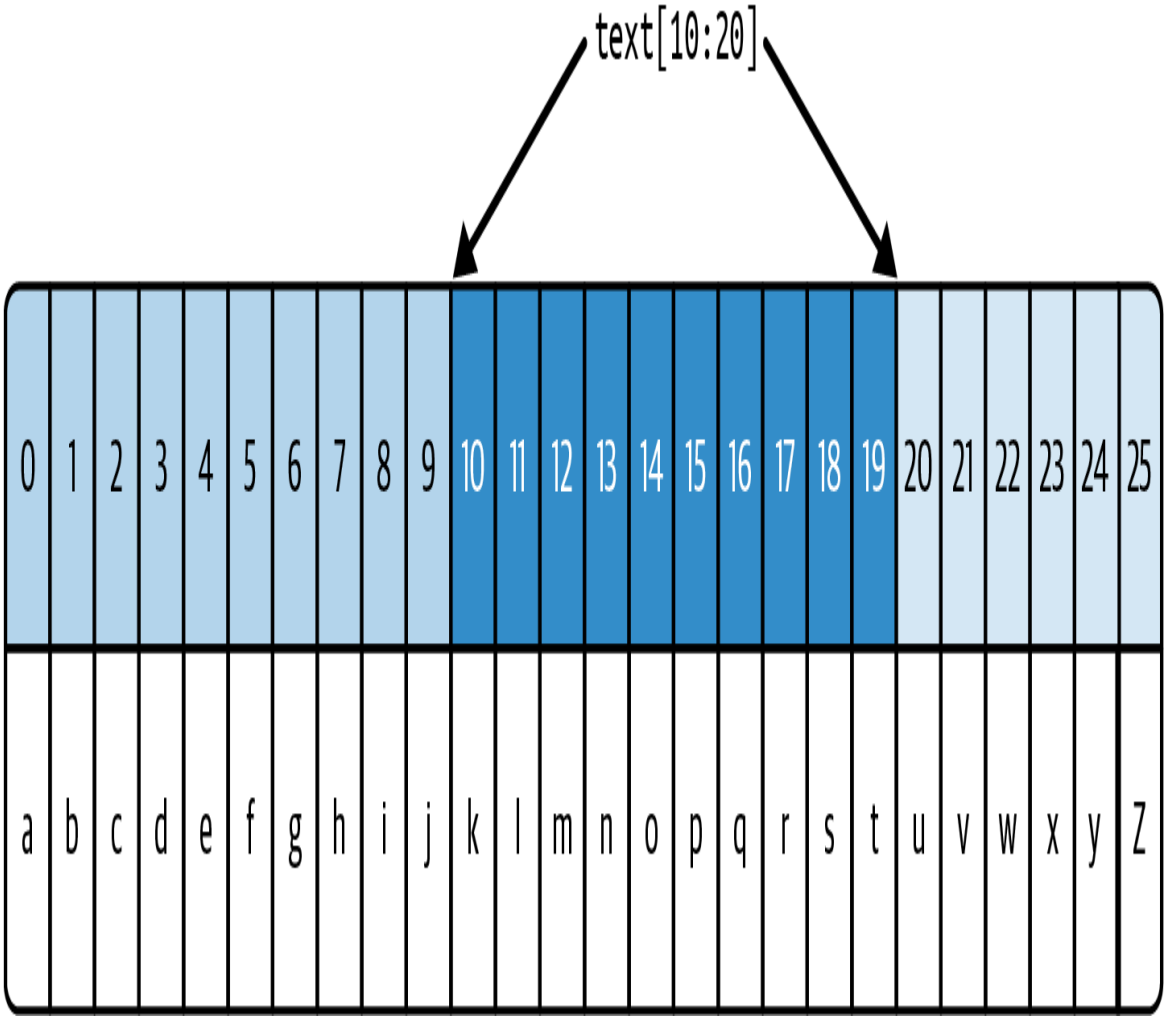


Figure 4-2. How to think about string slices

I will repeat that, because it's so important: slices, and many other ranges of numbers in Python, almost always operate up to and *not* including the endpoint. So if you ask for `10:20`, you will get back a string containing the characters in `text` starting from 10, up to and not including 20. In this case, that works out to be `'klmnopqrst'`.

What if you want all of the characters from the start of `text` through `'j'`? Well, `'j'` is the 10th letter of the alphabet, which means that it will be at index 9. Which means that you will need to use a slice from the start (i.e., index 0) until index 10 (since the slice's endpoint is never included):

```
text = 'abcdefghijklmnopqrstuvwxyz'
print(text[0:10])
```

The preceding code prints:

```
abcdefghij
```

A slice without a specified starting index begins at the start of the string, as you can see in Figure 4-3.

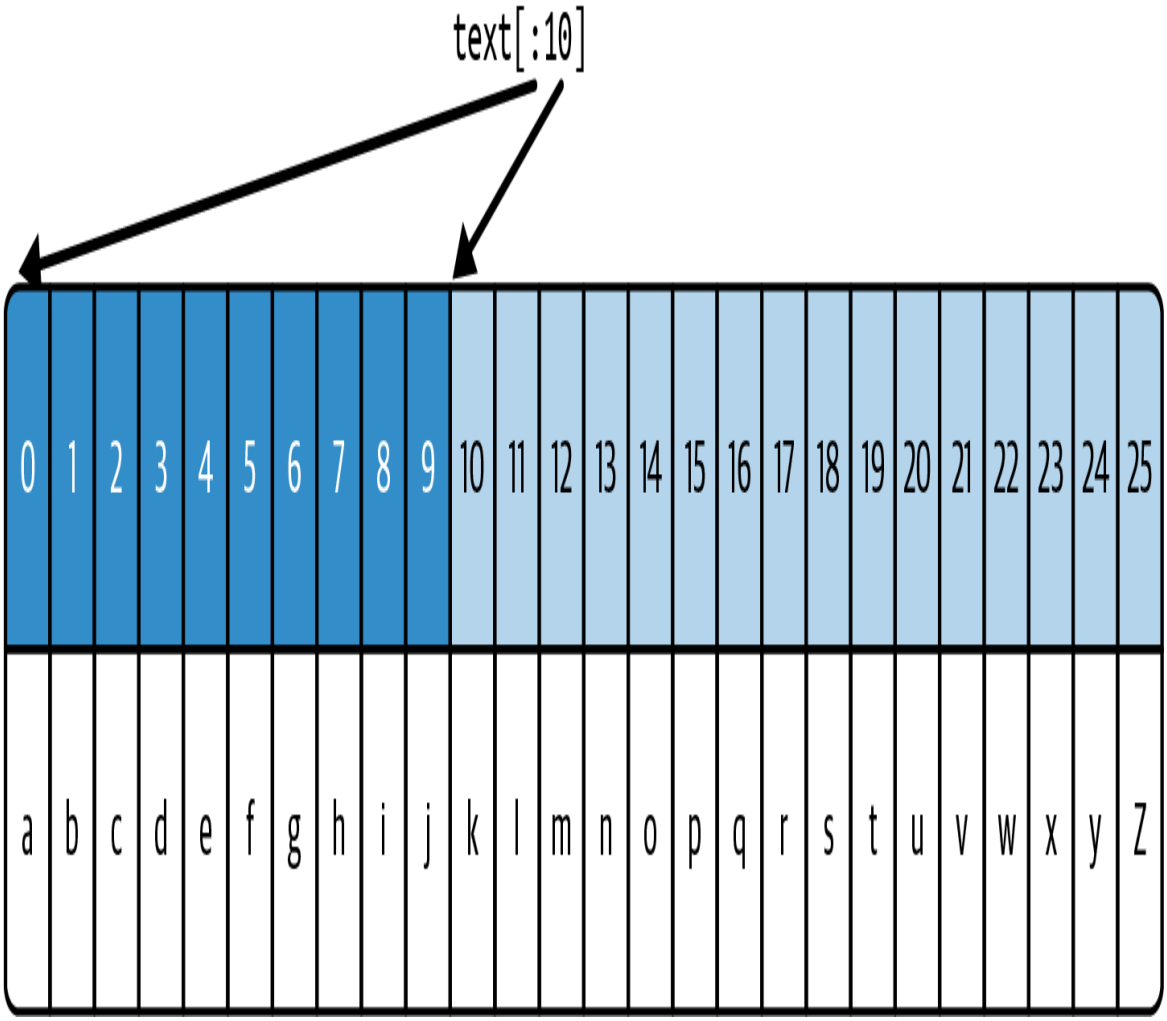


Figure 4-3. How to think about string slices with no start

If the slice starts with index 0, Python allows you to remove the index entirely:

```
text = 'abcdefghijklmnopqrstuvwxyz'
print(text[:10])
```

The above code prints:

```
abcdefghij
```

You can similarly remove the ending index, if you want to get through the end of the string:

```
text = 'abcdefghijklmnopqrstuvwxyz'
print(text[20:])
```

The preceding code prints `'uvwxyz'`, since `'u'` is at index 20 in `text`. By leaving out the ending index, I effectively asked Python to return through the end of the string. A slice without an ending index goes through the end of the string, as you can see in Figure 4-4.

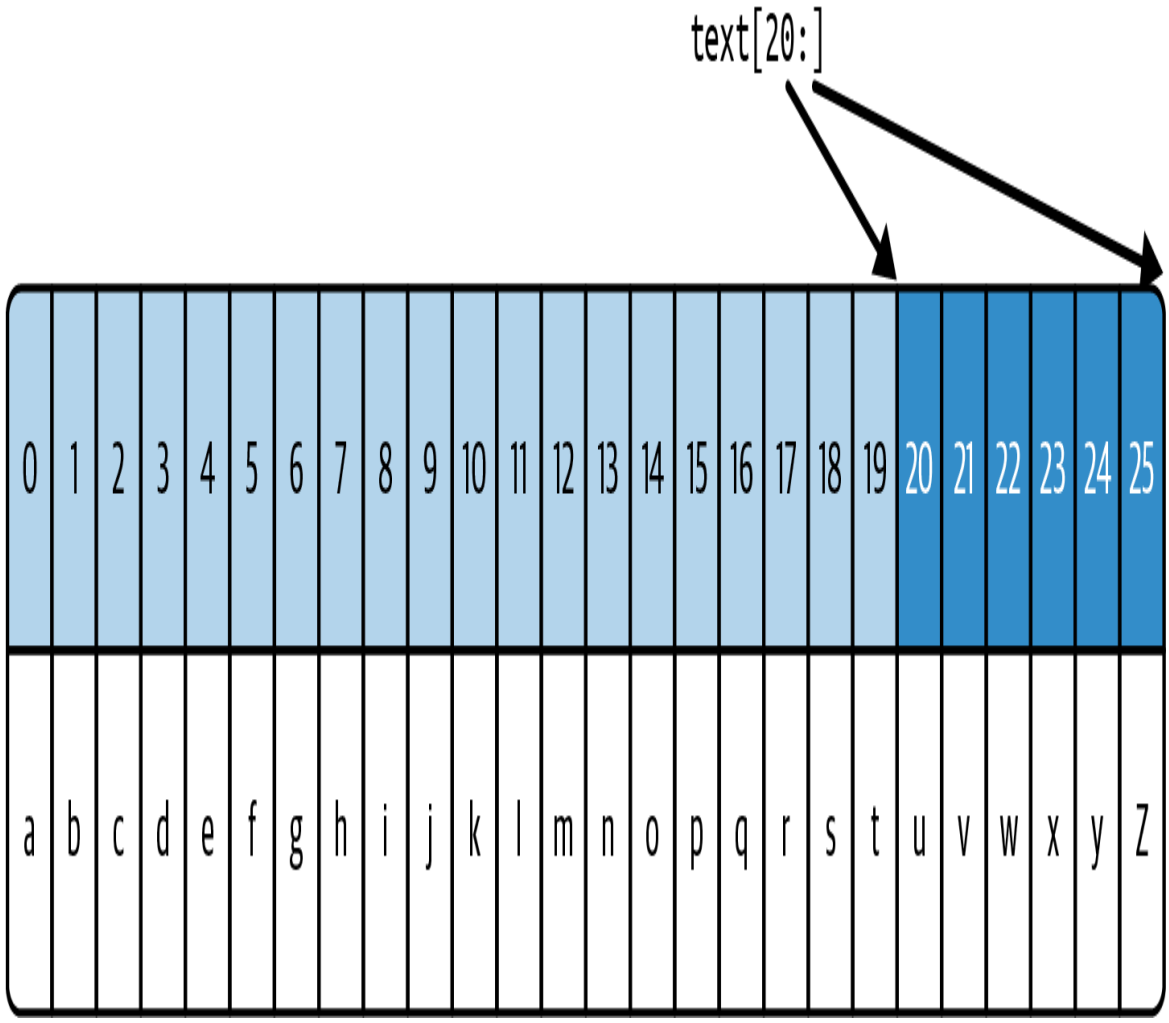


Figure 4-4. How to think about string slices with no end

Looking in a String

Python's `in` operator allows us to search in a string for one or more characters. It returns `True` if the string on the left can be found in the string on the right. For example:

```
text = 'abcdefghijklmnopqrstuvwxyz'
print('j' in text)      # True
print('!' in text)      # False
print('jkl' in text)    # True
print('aeiou' in text) # False
```

In the preceding example, notice that `'jkl'` is found in the alphabet, because that sequence of three characters appears, in that order and all together, in `text`. By contrast, all five characters in `'aeiou'` appear in `text`, but they do not appear together. `in` thus returns `False`.

Consider a program that asks the user to enter a lowercase word, and then says whether the word begins with a vowel. One way to write it is:

```
word = input('Enter a word: ')

if (word[0] == 'a' or word[0] == 'e' or
    word[0] == 'i' or word[0] == 'o' or word[0] == 'u'):
    print(f'Yes, {word} starts with a vowel')
else:
    print(f'No, {word} does *not* start with a vowel')
```

The preceding code works, but the `if` condition is a bit long. Using `in`, the condition can be easier to both write and read, and even more efficient for Python to execute:

```
word = input('Enter a word: ')

if word[0] in 'aeiou':
    print(f'Yes, {word} starts with a vowel')
else:
    print(f'No, {word} does *not* start with a vowel')
```

Special Characters

I mentioned earlier that a string can contain any number and any combination of characters. However, things get tricky if you want a string to contain more than a single line. That's because Python sees the end of a line on the screen as the end of a line of code.

Consider what happens if you try to define a string that contains multiple lines:

```
text = 'abcd
efgh'
```

Python will respond with an error message:

```
    text = 'abcd
           ^
SyntaxError: unterminated string literal (detected at line 1)
```

Python is complaining that the string started on the first line was never closed (i.e., it was "unterminated"). You might argue that it was closed on the second line, but Python generally sees the end of the line as the end of a statement or expression. In this case, it never checked line 2 to see if the string was closed.

To include multiple lines inside a string, you will need to use the special sequence `'\n'`. This two-character sequence is interpreted by Python as a "newline," meaning that when the string is printed, it'll move the cursor down one row. For example:

```
text = 'abcd\nefgh'
print(text)
```

The result of running the preceding code is:

```
abcd
efgh
```

While you type two characters to enter a newline, the sequence is interpreted by Python as a single character:

```
text = 'abcd\nefgh'
print(len(text))
```

The preceding code prints 9 on the screen. This demonstrates that while whitespace might not look like a character (because you cannot see it), it is still a character and needs to be included in your calculations.

`'\n'` is one of several special sequences that you can use in a string, all of which start with the backslash, `\`. For example, you previously saw that you can enter a literal quote with `'\"'` or `"\'"`.

There are a number of other special characters. The most useful of them, in my work, is tab, which is written as `'\t'`, which adds as much space as is needed to get the text to the next multiple-of-8 column. I often use it to create simple tables, assuming that the text is all of relatively uniform length. For a full list of escape sequences, you can look at the Python documentation.

What if you want to enter a literal `'\'` character? Then you enter two backslashes in a row, `'\\'`, which is interpreted as a single backslash.

Special Quotes

You have already seen four ways to create strings:

- Using `''`, single quotes
- Using `""`, double quotes
- f-strings (with either single or double quotes)
- Invoking `str` on an existing value, which returns a new string

There are a few other ways to create strings that are worth knowing about.

For example, if you use Windows, then you might sometimes need to assign a filename to a variable. If that filename contains folder names, what is known as a "full path," then it might contain some backslash characters, as well. For example:

```
path = 'c:\abcd\ef\ghij\klmn.txt'
print(path)
```

This will work, but it probably won't print what you expect. As you can see, the leading `'\a'` has been removed:

```
c:bcd\ef\ghij\klmn.txt
```

Moreover, modern versions of Python produce a warning — not an error that stops the program, but a message indicating that you're likely doing something wrong:

```
SyntaxWarning: "\e" is an invalid escape sequence. Such sequences
will
not work in the future. Did you mean "\\e"? A raw string is also
an
option.
```

What is going on here?

Simply put, Python recognizes certain escape sequences, such as `'\n'` and `'\t'`, as valid. It used to silently ignore any invalid sequences, such as `'\e'`, but as of several years ago, Python will warn you that you can't just put a backslash before any character and expect things to work correctly. Having `'\e'`, `'\g'`, and `'\k'` in the string led to this warning.

By contrast, `'\a'` *is* a recognized escape sequence, one that makes a sound technically known as the *alarm bell*. (On some computers, printing the string as I gave you will indeed produce a short "ding!" sound.)

To solve both of these problems in the preceding program, you will want to double all of the backslashes. Adding a backslash before `'\a'` will ensure that it is printed as intended. And adding a backslash before `'\e'`, `'\g'`, and `'\k'` will tell Python that you don't believe that any of these should be special escape sequences:

```
path = 'c:\\abcd\\ef\\ghij\\klmn.txt'
print(path)
```

Running the preceding code produces the intended result, displaying

```
c:\abcd\ef\ghij\klmn.txt
```

without any errors or warnings. However, doubling all of the backslashes in this way is a bit of a pain.

Fortunately, Python offers us a solution, "raw strings." A raw string looks like a regular string, created with either single or double quotes, but with the letter `r` before the opening quote. In a raw string, all of the backslashes are doubled, removing any escaped characters. For example:

```
path = r'c:\abcd\ef\ghij\klmn.txt'
print(path)
```

I tend to use raw strings either when writing Windows paths (as in this example) or when using regular expressions, a system for describing patterns of text whose syntax includes many backslashes. In such cases, it's quite convenient to avoid manually doubling all of the backslashes.

Finally, you saw earlier that we can include `'\n'` in our string to include a newline. That's fine for small pieces of text, but creating a full-length article, log message, or web page on a single line with `'\n'` every time we want to have a new paragraph is a bit hard to handle.

For this reason, Python offers "triple-quoted strings." Open a string with `'''` or `"""` (i.e., three quote characters in a row), and you can include any text you want, including a literal newline. In other words, triple-quoted strings can be spread across more than one line, allowing you to write more natural-looking text inside your code. For example:

```
text = """This is the first sentence.
This is the second.
And this is the third!
"""

print(text)
```

Sure enough, running the preceding code gives the following output:

```
This is the first sentence.
This is the second.
And this is the third!
```

Triple-quoted strings are a great example of "syntactic sugar," a term that means, "an easier-to-read, easier-to-write alternative to the standard syntax." Any newline included in a triple-quoted string is just converted into a `'\n'` character behind the scenes. Triple-quoted strings are used not only in defining long, multiline blocks of text, but also in documenting functions and modules, topics you will learn about in later chapters.

Strings Are Immutable

Let's say you create a string:

```
text = 'abcd'
```

You've seen that you can retrieve the first character with `text[0]`. But what if you want to change the string? What happens if you assign a new value to `text[0]`?

```
text[0] = '!'
```

You will get an error message:

```
    text[0] = '!'
       ~^^^
TypeError: 'str' object does not support item assignment
```

Python tells you that you cannot assign to an individual item within a string. This is true, and somewhat surprising. In many other programming languages, it's natural to not only retrieve from strings, but to also replace one or more characters within that string.

However, strings in Python are *immutable*. This means that a string, once defined, cannot be changed. Its length and contents are defined at creation time, and cannot be modified.

It might seem strange that Python, which is often used to process and manipulate text, has immutable strings. However, keeping them immutable

makes them more efficient, and also allows them to be used as dictionary keys, as you will see in Chapter 7. However, it does mean that you can only "modify" a string by retrieving parts of it and then reassembling it into a new string.

Immutability is a hard concept for many people to understand, so don't feel bad if you don't get it right away.

Integers are also immutable, but it's easy to understand how that works: if I assign `n = 5`, then I cannot say that the `5` that `n` refers to should really be worth `6` or `7`. I assigned `5` to `n`, and thus `n` will continue to refer to `5` until I assign it another, different value.

The same is true with strings: if I assign `text = 'abcd'`, then the value `'abcd'` cannot change. I can, however, assign a new, different value to `text`.

If this seems confusing, then you might be mixing up the concept of a "constant" with that of "immutable." Constants, which don't exist in Python, are like variables, except that they can only be assigned to once. Trying to assign a new, different value to a constant will result in an error. But again, Python does not have constants; it's always possible to assign a value to a variable. Immutability, by contrast, means that you cannot change a value.

The bottom line, at least for now, is that once you have created a string in Python, you cannot change it. And don't be surprised when you get an error message if you try to make such a change.

String Methods

You have already learned about, and used `print`, `input`, and `len`. These are all functions, which I often describe as the verbs of a programming language. This is true, but Python has another type of verb, as well, known as a *method*. Whereas functions are not inherently connected to any data structure, methods are defined within the context of such a data structure. Methods are actually far more common than functions in Python. As you

learn more Python, you will learn about more data structures and the methods that they support.

To invoke a function, you give its name, followed by `()`, putting any arguments — values you want to pass to the function — inside the parentheses:

```
a_function(arg1, arg2)
```

By contrast, a method will always be invoked via a data structure. You first give the data structure, then a `.` character, then the method name — followed by `()`, with any additional arguments inside:

```
data.a_method(arg1, arg2)
```

> **NOTE**
>
> Methods come from the world of *object-oriented programming*, a style that I don't cover in this book, but its influence is felt throughout the Python language. When you program using objects, you define data structures along with their functions, known as "methods." Keeping the definitions together can make for easier-to-understand, more maintainable code. For now, it's enough to understand that each method is defined to work with one particular data structure. Strings thus have "string methods." When you learn about lists in Chapter 6, and when you learn about dicts in Chapter 7, you will learn about dict methods. This is true for the built-in types that we'll cover in this book, and also for user-defined types used in more advanced programs.

str.strip

I'm introducing methods at this point in the book because strings support many different, useful methods. For example, consider this code:

```
name = input('Enter your name: ')
print(f'Hello, {name}!')
```

There is nothing inherently wrong with the preceding code. When run, it asks for the user's name. The user enters a name, and whatever the user entered is then printed as part of the greeting.

But what if the user accidentally presses the space bar before and after entering their name? You have already seen that spaces are characters, and are recorded in strings along with all other characters. This might lead to a strange-looking sort of greeting, such as:

```
Hello,    Reuven     !
```

This might seem like a silly or trivial example, but it's actually quite important. Consider, for example, when you enter username and password into an application. If there are spaces before or after your username or password, then you likely won't be able to log in — and you will likely be confused as to what has gone wrong.

Fortunately, Python provides the `str.strip` method. From its name, you can see that it's a string method, meaning that it only works on strings. When invoked, `str.strip` returns a new string, one without any leading or trailing whitespace characters. (Whitespace in the middle of the string is untouched.) For example:

```
name = input('Enter your name: ')
name = name.strip()
print(f'Hello, {name}!')
```

The first line of the preceding program asks the user to enter their name, as usual. The second line invokes `str.strip` on the string stored in `name`, expressed here as `name.strip`. The method returns whatever was in `name`, minus any whitespace at the beginning or end of the string. The return value from `name.strip` is then assigned back to `name`. By the time `print` displays the greeting, there is no trace of the whitespace. Even if the user entered many spaces before and after their name, they'll still see something like this:

```
Hello, Reuven!
```

Notice that `str.strip` returns a new string, rather than modifying the string on which it is invoked. But of course, that has to be the case, because

strings are immutable — they cannot be changed. However, many string methods (including `str.strip`) will return a new string that we will likely want to use in place of the original.

The preceding code works, but can be simplified a bit. Instead of taking the return value from `input`, a string, and assigning it to `name`, you could invoke `str.strip` directly on that (anonymous) string, then assigning the method's output to `name`:

```
name = input('Enter your name: ').strip()
print(f'Hello, {name}!')
```

The preceding code has the same result as the previous example. However, it only assigns to `name` once, with the result of `str.strip`, which is itself invoked on the result from `input`.

You can learn more about `str.strip` via the online Python documentation.

Case-Related Methods

You have already seen that Python strings are case sensitive, meaning that `'a'` and `'A'` are seen as completely different characters. If the user enters `'Abcd'` and you are looking for `'abcd'`, then they will not be seen as equal, which can throw off your `if` statement. For this reason, Python offers a number of string methods having to do with case. Each method returns a new string, based on the original one, with the case modified. Table 4-2 lists these methods. In each example, assume that `text` is the string `'aBcD eFgH'`.

Table 4-2. Case-related string methods

Method	Description	Example	Result
`str.lower`	All letters are lowercased	`s.lower()`	`'abcd efgh'`
`str.upper`	All letters are capitalized	`s.upper()`	`'ABCD EFGH'`
`str.capitalize`	First letter is capitalized, others lowercased	`s.capitalize()`	`'Abcd efgh'`
`str.title`	First letter of each word is capitalized, others lowercased	`s.title()`	`'Abcd Efgh'`
`str.casefold`	All letters are lowercased, handling nonEnglish characters, too	`s.casefold()`	`'abcd efgh'`
`str.swapcase`	Each letter's case is switched	`s.swapcase()`	`'AbCd EfGh'`

Using these methods, you can compare a user's input with the string you were expecting, without having to worry about matching the case. For example:

```
secret = 'shhh'

word = input('Enter secret: ').strip()
```

```
if secret == word.lower():
    print('You got it!')
else:
    print('Nope!')
```

With the preceding code, the user can enter *Shhh*, *shhh*, *SHHH*, or any other combination of capital and lowercase letters, and it'll still work. Because of the `str.strip` on the result from `input`, there can even be spaces before or after the word.

If you do not need to preserve the user's original input, you could even invoke `str.lower` on the output from `str.strip`:

```
secret = 'shhh'

word = input('Enter secret: ').strip().lower()

if secret == word:
    print('You got it!')
else:
    print('Nope!')
```

Invoking multiple methods in this way is known as "method chaining." It works because `input` returns a string, on which you invoke `str.strip`. That returns a string, on which you invoke the `str.lower` method. `str.lower` returns a string, which is assigned to `word`.

NOTE

Some languages have more than one way to write particular characters. For example, the German "ß" can also be written as "ss". If you use `str.lower` to compare them, they will be seen as distinct. That's where Python's `str.casefold` method comes in: it takes non-English comparison rules into account. For example:

```
s1 = 'straße'
s2 = 'strasse'

print(s1 == s2)                          # False
print(s1.lower() == s2.lower())          # False
print(s1.casefold() == s2.casefold())    # True
```

You can learn more about `str.lower` and other case-related methods in the online Python documentation.

Number-Testing Methods

Earlier in this chapter, you saw that you can get an integer from a string by invoking `int`. For example:

```
text = input('Enter a number: ').strip()
n = int(text)

print(n * 2)
```

If the user enters `1234` in response to the preceding program, then `int` will create a new integer based on `'1234'`, which is then assigned to `n`. The program will then print `2468`.

But what if the user enters a string that cannot be turned into an integer? For example, if the user enters `hello` in response to the prompt? Python will try to convert `text` into an integer, and will produce the following error message:

```
    n = int(text)
ValueError: invalid literal for int() with base 10: 'hello'
```

Because `text` contains characters other than digits, `int` cannot turn it into an integer, and instead raises a `ValueError` exception, which stops the program. It would be nice to identify that `text` cannot be turned into an integer, and indicate this before the error occurs.

The method `str.isdecimal` can be used for such purposes. `str.isdecimal` returns `True` if a string contains only the digits 0-9. Otherwise, it returns `False`. (There are related methods called `str.isdigit` and `str.isnumeric`, and you will get nearly identical results from them. In this book, I will stick with `str.isdecimal`.) The one big drawback of `str.isdecimal` is that it cannot handle negative numbers, because `'-'` is not a decimal digit.

Here is a version of the preceding program that no longer produces an exception if the user enters non-digits:

```
text = input('Enter a number: ').strip()

if text.isdecimal():
    n = int(text)
    print(n * 2)
else:
    print(f'{text} is not numeric; ignoring')
```

Notice that `str.isdecimal` is a *string* method. It tells us whether a string can be handed to `int`. It is not a method that you run on an integer, despite the "digit" in its name.

Also remember that the point of invoking `str.isdecimal` in this program is to avoid invoking `int` on a string that contains non-digits. For this reason, the call to `str.isdecimal` must come as part of the `if`, before the invocation of `int`.

You can learn more about `str.isdecimal` via the online Python documentation.

Exercise: Display a Character

In this exercise, you will ask the user to enter a word. You will also ask the user to enter an index. Assuming the index is numeric and within the word's boundaries, display the index, word, and character.

Specifications

- Ask the user to enter a word.
- Ask the user to enter a numeric index.
- If the index is non-numeric, give the user an error message.
- If the index is less than 0, give the user an error message.
- If the index is beyond the word's bounds, give the user an error message.
- Assuming all is well, print the index, word, and character.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

My solution starts by getting two separate inputs from the user, which I assign to two variables, `word` and `index_text`:

```
word = input('Enter a word: ').strip()
index_text = input('Enter an index: ').strip()
```

I could have just called the second variable `index` and then assigned an integer to it. However, I often like to separate variables that will contain different types.

Next, I used `str.isdecimal` to check whether I can even turn `index_text` into an integer. Notice that only after I see that `str.isdecimal` returns `True` do I invoke `int(index_text)`, assigning it to `index`. If the user's input is non-numeric, then I print an error message to the user in the `else` block:

```
if index_text.isdecimal():
    index = int(index_text)

else:
    print(f'Index {index_text} is not numeric; ignoring')
```

Finally, inside the `if` block, I have a second, inner `if`/`elif`/`else` block, checking whether the index that the user provided is too low or too high:

```
word = input('Enter a word: ').strip()
index_text = input('Enter an index: ').strip()

if index_text.isdecimal():
    index = int(index_text)

    if index < 0:
        print('Index is negative; ignoring')
    elif index >= len(word):
        print('Index is too high; ignoring')
    else:
        print(f'Index {index} in {word} is {word[index]}')
else:
    print(f'Index {index_text} is not numeric; ignoring')
```

From Python's perspective, there is nothing wrong with having an `if` statement in the block of another `if` statement. Notice that the blocks inside the inner `if` statement are all indented four additional spaces. This is typical of Python, in that you can have indented blocks inside of indented blocks.

Also notice the comparison in the `elif` statement. It does not compare `index > len(word)`, because that would ignore the case in which `index` is the same as `len(word)`, which would be one beyond the maximum index in `word`. Instead, I used `index >= len(word)`, which ensures that we avoid that problem.

GenAI Task: Reverse Socratic Method

For this exercise, I want you to explore using the "Reverse Socratic Method" technique. Basically, you should ask a chatbot lots of questions about the code, how it is written, why it is written in this way, and how you could change it. Your job here is to ask as many questions as you can until you feel comfortable not just with the code, but with what it does and how it works.

The Reverse Socratic Method is how I learn and explore many new topics in my own learning. However, I'm all too aware of the fact that chatbots make mistakes, often big ones. It's therefore important not to accept what the chatbot says, but to challenge it and argue with it. Not only will such challenges help to solidify your understanding, but they'll also act as a check on the chatbot. If it gives contradictory responses, or says things that don't make sense, you can keep challenging and asking it.

Here is a short exchange that I had with ChatGPT about this program. (The GenAI tasks in this book switch between ChatGPT, Claude, and Gemini.)

Prompt:

```
I'm reading an introductory Python book, "Python for non-
programmers."
I have just finished chapter 4 about numbers and strings, and
wrote
the following program:

word = input('Enter a word: ').strip()
index_text = input('Enter an index: ').strip()

if index_text.isdecimal():
    index = int(index_text)

    if index < 0:
        print('Index is negative; ignoring')
    elif index >= len(word):
        print('Index is too high; ignoring')
    else:
        print(f'Index {index} in {word} is {word[index]}')
else:
    print(f'Index {index_text} is not numeric; ignoring')

I want to ask you questions about this program to ensure that I
```

understand what is happening, and to explore ideas that it introduces
and that I don't quite understand.

ChatGPT:

Perfect — that's a great idea. This program is a nice little example
that combines string handling, input validation, type conversion, and
indexing — all key ideas from early Python chapters.

Go ahead and ask your questions — we can go line by line, or explore
any concept (like isdigit, len, string indexing, or strip) in as much
depth as you like. What would you like to start with?

Prompt:

It's a bit frustrating to me that even though Python supports negative
indexes on strings, str.isdecimal does not. I thus cannot count from the
right side. I'm still a Python beginner, and don't want to get into
complex topics such as exception handling. Is there a way, using
str.isdecimal, to handle negative numbers? Or should I leave that for
later in my Python learning?

ChatGPT:

That's a really insightful observation — and it shows you're already
thinking like a programmer.

You're absolutely right:

str.isdecimal() returns True only if all characters in the string are
digits (0–9).

So '3' → ✅ True, but '-3' → ❌ False, because of the minus sign.

Python's philosophy here is that isdigit() checks only for digits, not
for numeric signs or formatting.

✅ Option 1: The simple, beginner-friendly approach

Since you don't want to use try/except yet, the easiest fix is to manually check for a leading '-', and then check the rest of the string with .isdecimal().

Here's how you could do that:

```
    if index_text.isdecimal() or (index_text.startswith('-') and
    index_text[1:].isdecimal()):
        index = int(index_text)
        # continue as before
    else:
        print(f'Index {index_text} is not numeric; ignoring')
```

Explanation:

If the string starts with '-', we skip that first character using index_text[1:] and check if the rest are digits.

This way, '5' and '-5' both count as numeric.

This adds only one small condition, and doesn't require exception handling or new concepts.

Prompt:

Why use str.startswith? (I haven't learned about that, but I understand the meaning.) Why not just say index_text[0] == '-'?

ChatGPT:

Excellent question — and it shows you're thinking carefully about how string operations work, not just copying examples.

You're right:
You could absolutely write

```
index_text[0] == '-'
```

```
and it would work most of the time.

The key difference between that and .startswith('-') is how they
behave with empty strings.
```

Summary and Conclusion

In this chapter, you learned about two of the most important kinds of data in Python, numbers (`int` and `float`) and text (`str`). You will use these types in every program you write, either directly or indirectly. Along the way, you also learned:

- The differences between the numeric `int` and `float` types
- Numeric operators
- The use of += to increment a number
- What strings are, and how to define them
- Triple-quoted and raw strings
- Retrieving characters and slices from a string
- What it means for strings to be "immutable"
- Some basic string methods, and how to use them

In the next chapter, we'll see how to manipulate strings using loops — and then we'll move on to lists and tuples, the two other members of the "sequence" family of data structures.

Chapter 5. Repeating with Loops

Computers can seem magical, or even brilliant. But really, that's just an illusion: computers are not very smart, but they are very good at following directions. Moreover, they can do so very quickly, performing billions of operations each second. By contrast, humans can be quite smart, but we're very slow. Plus, we get bored or tired when forced to repeat a task.

In many ways, programming is about offloading repetitive, boring work to a computer, taking advantage of its patience and speed. It's so common to write such programs that every programming language has special constructs for doing so, known as *loops*. In a loop, you tell the computer to repeat some code a number of times, or until some condition has been met.

In this chapter, you'll learn about Python's two loop constructs, `for` and `while`. Along the way, you'll learn about *ranges*, *indexes*, and how to exit early from a loop. After finishing this chapter, you'll understand why loops are such a central part of programming, and how they can help you concentrate on the big picture, leaving the boring work to the computer.

DRY: Don't Repeat Yourself

Let's say that you have assigned the string `'abcd'` to the variable `text`, and want to print each of the characters in `text` on a separate line. You could write:

```
text = 'abcd'

print(text[0])
print(text[1])
print(text[2])
print(text[3])
```

This code works — or as I often like to say in my classes, "Unfortunately, this works." Because it does get the job done, but it does so in a long, roundabout way. These four lines of code are almost identical to one another, which suggests that there's a way to crunch it down into something smaller.

Avoiding repetition in your code is best summed up by the acronym *DRY*, short for "don't repeat yourself." If you find yourself writing identical, or near-identical, code on several consecutive lines, then you have almost certainly identified an opportunity to "DRY up" your code.

Why is it generally a good idea to avoid repetition in your code?

First and foremost, because programmers spend most of their time maintaining and debugging existing code, not writing new code. By avoiding repetition, you reduce the amount of code you'll need to maintain. This is true if you already know the code, but it's even more important when you start working on an unfamiliar program. Wading through pages of repeated code is difficult and time-consuming.

A second, related reason is that coding requires keeping a great deal of information in your head at the same time. Shorter programs are easier to remember and understand, and occupy less of your mental capacity than longer programs.

Also, it's almost inevitable that you will have to change your code in the future — to make it run faster, to fix bugs, or to change the functionality. By avoiding repetition, you cut down on the number of times you need to make the same correction.

for Loops

One key tool that helps to avoid repetition is a *loop*, which repeats a block of code a number of times. `for` loops are the most common type you'll encounter and use in Python; later in this chapter, you'll also learn about `while` loops. In a `for` loop, you tell Python that you want to repeat the

same code once for every item. For example, if you want to do something for every character in a string, a `for` loop fits perfectly.

> **NOTE**
>
> Data that knows how to behave inside a `for` loop is known as *iterable* in Python. Because you can put a string inside of a `for` loop, you can say that "strings are iterable." You'll learn about other iterable types later in this book. Similarly, each pass through the loop is known as an *iteration*.

Here is an example `for` loop that does precisely what the earlier code block did, but more compactly and elegantly:

```
text = 'abcd'

for one_character in text:
    print(one_character)
```

Here are some things to notice about the syntax of a `for` loop:

- It starts with the word `for`.
- Following the `for` comes the *loop variable*. This is the variable that will get a new value with each loop iteration. As with all variables, the name should be meaningful to you, so that you can better understand and maintain your code. Python's behavior will not be affected by the variable's name.
- After the loop variable comes the word `in`. Here, `in` is not being used for searching in a string. It's just part of the `for` loop's syntax.
- Next comes the iterable value on which Python should iterate. This can be either an actual value or a variable referring to such a value.
- The line ends with a `:`, much as you've already seen with `if` and `else`.

- Finally, you write the *loop body* — an indented block of code that will execute repeatedly, once per iteration.
    - In each iteration, the loop variable will have a different value.
    - The loop body can contain any Python code you want, including `print`, `input`, `if`, and even an inner (*nested*) `for` loop.

> **NOTE**
>
> Many programmers give loop variables short names, such as `i` or `c`. I prefer to go in the other direction, naming loop variables starting with `one_` to make it clear what type of value will be assigned in each iteration. So if we're getting one character at a time, I'll use `one_character` as my loop variable.

That explains the syntax of a `for` loop. But in many ways, it's easiest to understand the loop as a conversation:

1. Python turns to `text`, and asks: *Are you iterable*?
2. `text` responds by saying, *Yes, I am iterable*. (If it isn't iterable, then the loop exits with an error.)
3. The loop asks `text` for its next value.
4. If `text` has no more values to offer, then the loop ends.
5. The value returned by `text` is assigned to the loop variable, `one_character`.
6. The indented block executes.
7. Execution returns to step 3.

You might prefer to think about that `for` loop using a table, considering what happens with each iteration (see Table 5-1).

Table 5-1. What happens in a `for` loop?

Iteration	`one_character`	Printed output
Before loop	—	—
1st	`'a'`	a
2nd	`'b'`	a b
3rd	`'c'`	a b c
4th	`'d'`	a b c d
Loop ends	`'d'`	a b c d

Exercise: Vowels, Digits, and Others

In this exercise, you will ask the user to enter some text. You'll then use a `for` loop to go through the characters in that string, counting the number of vowels, the number of digits, and the number of others.

Specifications

Here's what you need to do:

- Define three variables, `vowels`, `digits`, and `others`, assigning them all the integer 0.
- Ask the user to enter some text, and assign it to the variable `text`.
- Go through the string, one character at a time.

  - Each time you encounter a vowel (a, e, i, o, or u), add 1 to `vowels`.
  - Each time you encounter a digit (0-9), add 1 to `digits`.
  - In all other cases, add 1 to `others`. This includes punctuation and whitespace characters.
- Print the values of `vowels`, `digits`, and `others` on the screen.

Example:

```
Enter text: hello!! 123

vowels: 2
digits: 3
others: 6
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The program starts by defining the three counting variables, as well as the user's input:

```
vowels = 0
digits = 0
others = 0

text = input('Enter text: ').strip()
```

Next, the program goes through each character with a `for` loop. Inside that loop, I used `if`:

- To check if the character is a vowel, I check `one_character in 'aeiou'`, searching inside the string with `in`.
- To check if the character is a digit, I invoke `one_character.isdecimal()`, invoking the

`str.isdecimal` method.

- If neither of the preceding conditions applies, the `else` block adds 1 to `others`.

Here is the code:

```
for one_character in text:
    if one_character in 'aeiou':
        vowels += 1
    elif one_character.isdecimal():
        digits += 1
    else:
        others += 1
```

Notice that the string containing vowels only contains them in lowercase form. To count capitalized vowels, invoke `str.lower` on `text`, and iterate over the resulting string:

```
for one_character in text.lower():
    if one_character in 'aeiou':
        vowels += 1
    elif one_character.isdecimal():
        digits += 1
    else:
        others += 1
```

Finally, the program prints the results:

```
print(f'vowels = {vowels}')
print(f'digits = {digits}')
print(f'others = {others}')
```

If you are using f-strings to print both a variable and its value, as seen in the preceding code, you can actually take advantage of a cool trick: put an = inside the braces, and both the name and value will be displayed for you, separated by an = sign:

```
print(f'{vowels=}')
print(f'{digits=}')
```

```
print(f'{others=}')
```

The full solution can thus look like this:

```
vowels = 0
digits = 0
others = 0

text = input('Enter text: ').strip()

for one_character in text.lower():
    if one_character in 'aeiou':
        vowels += 1
    elif one_character.isdecimal():
        digits += 1
    else:
        others += 1

print(f'{vowels=}')
print(f'{digits=}')
print(f'{others=}')
```

Table 5-2 describes a run of this final version of the code, assuming that the input text is `'hi!! 12'`.

Table 5-2. What happens in each iteration?

Iteration	`one_character`	`vowels`	`digits`	`other`
Before loop	—	0	0	0
1st	`'h'`	0	0	1
2nd	`'i'`	1	0	1
3rd	`'!'`	1	0	2
4th	`'!'`	1	0	3
4th	`' '`	1	0	4
4th	`'1'`	1	1	4
4th	`'2'`	1	2	4
Loop ends	—	1	2	4

AI Challenge: Add a Bug

The AI challenge this time is similar to one you did in the previous chapter: give the AI chatbot a working version of the solution to this exercise. Ask it to introduce a bug. Your job is then to find the bug and explain what problems it causes to the program.

Prompt (with Gemini):

I'm reading an introductory Python book, "AI-assisted Python for non-programmers."

I have just finished chapter 5 about loops, and I just wrote the following program that counts the number of vowels, digits, and other
characters in a user's input text:

```
vowels = 0
digits = 0
others = 0

text = input('Enter text: ').strip()


for one_character in text.lower():
    if one_character in 'aeiou':
        vowels += 1
    elif one_character.isdecimal():
        digits += 1
    else:
        others += 1

print(f'{vowels=}')
print(f'{digits=}')
print(f'{others=}')
```

Let's play a game: Your job is to modify the program to introduce a
bug. My job is then to identify the bug and tell you why it'll cause
trouble. We should continue until I tell you to stop.

Gemini responded:

Great! I love this game - it's an excellent way to sharpen your debugging skills. Let me introduce the first bug:

```
vowels = 0
digits = 0
others = 0
text = input('Enter text: ').strip()
for one_character in text.lower():
    if one_character in 'aeiou':
```

```
        vowels += 1
    if one_character.isdecimal():
        digits += 1
    else:
        others += 1
print(f'{vowels=}')
print(f'{digits=}')
print(f'{others=}')
```

```
I've made one change to your program. Can you spot the bug and
explain
what problem it will cause?  Take your time to read through it
carefully and test it mentally with some example inputs! When
you're
ready, tell me what you found.
```

I wrote:

```
I see--instead of if/elif/else, I now have if/if/else. With
if/elif/else in place, one (and only one) block is guaranteed to
fire. But if if/if/else, the logic is changed. vowels will be
incremented if there is a vowel, and digits will be incremented
if
there's a digit. But others will be incremented for vowels, too,
because the "else" will fire whenever there is a non-digit.
```

Gemini wrote:

```
Excellent analysis! You nailed it!

You're absolutely right: with if/if/else, the else is only paired
with
the second if (the digit check), not the first one. So when the
program encounters a vowel:

The first if fires and increments vowels
The second if doesn't fire (vowels aren't digits)
The else fires and increments others

This means vowels get double-counted - once correctly as vowels,
and
once incorrectly as "others". The others count will be inflated
by the
number of vowels in the text!  Great job understanding the
control
```

```
flow! Ready for bug #2?

vowels = 0
digits = 0
others = 0
text = input('Enter text: ').strip()

for one_character in text:
    if one_character in 'aeiou':
        vowels += 1
    elif one_character.isdecimal():
        digits += 1
    else:
        others += 1

print(f'{vowels=}')
print(f'{digits=}')
print(f'{others=}')
```

Multiple Iterations

What if you want to perform a particular task several times? A `for` loop would seem to be the most natural way to do that. For example, you might consider:

```
for one_item in 3:
    print('Hooray!')
```

Although this code might seem reasonable and logical, it won't work:

```
Traceback (most recent call last):
  File "", line 1, in
    for one_item in 3:
                    ^
TypeError: 'int' object is not iterable
```

The first thing that a `for` loop does is ask the value at the end of the line whether it's iterable, meaning if it knows how to behave inside a `for` loop. Strings are iterable, as you've already seen. But it turns out that integers are not iterable. That's why the program exits with an error, with Python saying, "*int* object is not iterable."

There are many reasons why you might expect integers to be iterable. For one, `for` loops in other languages traditionally count up or down with integers. For another, iterating a set number of times seems like such a common thing that you would expect Python to include it. But no, as you see from this example, you cannot iterate over an integer.

Python does, however, provide the `range` built-in function, which makes it possible to achieve the same goal. Invoking `range(3)` returns a special `range` object, one which is designed to work inside `for` loops, providing three values:

```
for one_item in range(3):
    print('Hooray!')
```

In this case, `for` doesn't turn to the integer `3`, but rather to the value returned by `range(3)`, asking whether it's iterable. The answer is yes, and the loop body executes a total of three times:

```
Hooray!
Hooray!
Hooray!
```

But wait a second: a `for` loop doesn't just iterate a number of times. With each iteration, a new value is assigned to the loop variable — `one_item`, in this case. What value is being assigned to `one_item` in each iteration? Changing the code to include an f-string makes it clear:

```
for one_item in range(3):
    print(f'{one_item} Hooray!')
```

Here's the result:

```
0 Hooray!
1 Hooray!
2 Hooray!
```

In other words, looping over `range(3)` will indeed result in three iterations. But beyond that, with each iteration, `range(3)` returns an integer. It starts with 0, then returns 1, and then returns 2 before ending the loop.

Just as string indexes started with 0 and went up to (but not including) the string's length, `range(n)` starts counting with 0 and goes up to (but not including) the number passed to `range`. Iterating over `range(5)` will return the five integers 0, 1, 2, 3, and 4. And `range(100)` will return the 100 integers starting with 0, going up to (but not including) 100.

The value given to `range` must be an integer. But that integer can come from the user, not just from a hardcoded value. For example, this program asks the user how many numbers they want to sum. The program then asks them for that number of integers, adds them together, and prints the total:

```
text = input('How many numbers? ').strip()

if text.isdecimal():
    n = int(text)
    total = 0

    for index in range(n):
        text = input(f'Enter integer {index}: ').strip()

        if text.isdecimal():
            total += int(text)
            print(f'After adding {text}, total is {total}.')
        else:
            print(f'{text} is not numeric; ignoring.')

else:
    print(f'{text} is not numeric; too bad!')

print(f'Total = {total}')
```

The preceding program combines many of the lessons you have learned so far:

- The user enters a number, but `input` returns a string.

- If the user's input can be turned into an integer, having checked with `str.isdecimal()`, then the int value is assigned to `n`.
- The program then uses a `for` loop on `range(n)`, giving that many chances to add numbers together.
- Each input from the user is checked to see if it can be turned into an integer with `str.isdecimal()`.
- If the user's input can be turned into a digit, then it is, and the value is added to `total`.

Also notice the multiple levels of indentation in the preceding code. An `if` condition is in the first column, but then a `for` loop is in its block, and a second `if` is inside the body of the `for` loop.

Here is what a run of the preceding program looked like with four numbers — 2, 3, 10, and `'hello'`:

```
How many numbers? 4
Enter integer 0: 2
After adding 2, total is 2.
Enter integer 1: 3
After adding 3, total is 5.
Enter integer 2: 10
After adding 10, total is 15.
Enter integer 3: hello
hello is not numeric; ignoring.
Total = 15
```

What About the Index?

People coming to Python from other programming languages are often a bit puzzled by the way `for` loops work. That's because in other languages, `for` loops work with a numeric index, typically starting at 0 and going up to some maximum number. That's not so different from what you have seen with `range`, but it's quite different from how you would iterate over a string.

In C, for example, you cannot just ask a string to give you its characters, one at a time. Rather, it's more like:

1. Set `index` to 0.
2. Ask the string for its character at `index`.
3. Increment `index`.
4. If `index` is beyond the length of the string, stop the loop.
5. Go back to step 2.

The good news is that Python saves you from having to define, increment, and check the index. When you iterate over a string, there isn't any index to keep track of; you just work with the characters, which are supplied to you one at a time.

But there are times when you might want the index. For example, you might want to print each of the characters in a string, along with each character's index.

You could do this with `range`, emulating the way that C-like languages work:

```
text = 'abcd'

for index in range(len(text)):
    print(f'{index}: {text[index]}')
```

The preceding code does work, but it feels a bit odd to anyone with Python experience. Iterating over a `range` so that you can use the numbers as indexes into an already iterable sequence is almost never the right way to go.

Instead, you should retrieve the values in the sequence, incrementing the index with each iteration. For example:

```
text = 'abcd'
index = 0
```

```
for one_character in text:
    print(f'{index}: {one_character}')
    index += 1
```

Although this code works, it still feels a bit forced. Isn't there an easier, better way to get the index along with the number? Haven't a lot of other people wanted to do this before?

> **NOTE**
>
> Python is an open source language, developed by the community of coders who use it, not by a company looking to maximize profits. Generally speaking, this means that if a large number of people encounter a problem, the Python community will try to find a general solution, to make everyone's lives easier. When you encounter a problem, and there doesn't seem to be an easy, straightforward solution, check again: the odds are pretty good that someone else has encountered, and fixed, this problem before you.

There is an elegant solution to this problem: the built-in `enumerate` function. You invoke `enumerate` on an existing iterable value, such as a string. It then returns, with each iteration, *two* values — the index, starting with 0, and the value that would have been returned without using `enumerate`. Here's how it looks:

```
text = 'abcd'

for index, one_character in enumerate(text):
    print(f'{index}: {one_character}')
```

The preceding code is shorter and clearer than previous versions. Moreover, there are fewer opportunities for bugs, because `enumerate` is doing the hard work of tracking and incrementing `index`.

But the code does look weird, mostly because the `for` loop has *two* loop variables. That's right — with each iteration, `enumerate` returns *two* values, the index and the value from the original data structure. This seems weird, and you'll understand it more in Chapter 6, where you'll learn about

tuples and tuple unpacking. For now, it's enough to know that if you want to get the index, you can use `for` along with `enumerate`, and avoid working too hard. You can follow each iteration in the preceding `for` loop in Table 5-3.

Table 5-3. What happens in a `for` loop with `enumerate`?

Iteration	`index`	`one_character`	Printed output
Before loop	—	—	—
1st	0	`'a'`	a
2nd	1	`'b'`	a b
3rd	2	`'c'`	a b c
4th	3	`'d'`	a b c d
Loop ends	3	`'d'`	a b c d

Exercise: Expand Numbers

In this exercise, you'll ask the user for a number containing any number of digits, expanding it to show how each digit is the result of multiplying by a power of 10.

Specifications

Every decimal number can be expanded to be the sum of numbers multiplied by a power of 10. For example, the number 2468 can be rewritten as:

```
2 * 10 ** 3
4 * 10 ** 2
6 * 10 ** 1
8 * 10 ** 0
```

Notice that you can calculate each power of 10 using Python's `**` operator. Also notice that the ones digit is still multiplied by a power of 10, but that would be 10 to the zeroth power (i.e., `10 ** 0`), which is 1.

Your challenge:

1. Ask the user to enter a string containing only digits.
2. If the string contains any nondigits, give an error message and stop.
3. Go through each digit, printing it in the preceding format, with the digit multiplied by the appropriate power of 10.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

To start, ask the user to enter a string, and check that it contains only digits:

```
text = input('Enter a number: ').strip()

if text.isdecimal():
    print(f'Yes, {text} contains only digits')
else:
    print(f'Sorry, {text} contains non-digits; stopping.')
```

The heart of this program will be in the `if` block. A first pass through the program might look like this:

```
text = input('Enter a number: ').strip()

if text.isdecimal():
    for one_digit in text:
        print(f'{one_digit} * 10 ** n')
else:
    print(f'Sorry, {text} contains non-digits; stopping.')
```

This will work, at least in part — but it will be missing a big part of the program's intended output, namely the appropriate power of 10. In order to calculate the power of 10 in each iteration, you'll need to know where you are in the input string. Using `enumerate`, you can grab both the index and the current digit, and print them together:

```
text = input('Enter a number: ').strip()

if text.isdecimal():
    for index, one_digit in enumerate(text):
        print(f'{one_digit} * 10 ** {index}')
else:
    print(f'Sorry, {text} contains non-digits; stopping.')
```

The good news is that the program is closer to working than before. However, it's printing the exponents in reverse order; given the input `'2468'`, it needs to start with 3 and end with 0. That's not what happens:

```
Enter a number: 2468
2 * 10 ** 0
4 * 10 ** 1
6 * 10 ** 2
8 * 10 ** 3
```

Perhaps subtracting `index` from the length of the string would work:

```
text = input('Enter a number: ').strip()

if text.isdecimal():
    for index, one_digit in enumerate(text):
        print(f'{one_digit} * 10 ** {len(text) - index}')
else:
    print(f'Sorry, {text} contains non-digits; stopping.')
```

The good news is that this version is closer. However, it suffers from an *off-by-one error*, in that it isn't subtracting by quite enough:

```
Enter a number: 2468
2 * 10 ** 4
4 * 10 ** 3
```

```
6 * 10 ** 2
8 * 10 ** 1
```

This version subtracts `len(text) - index - 1`, ensuring that the numbers will end with 0 and start with `len(text) - 1`:

```
text = input('Enter a number: ').strip()

if text.isdecimal():
    for index, one_digit in enumerate(text):
        print(f'{one_digit} * 10 ** {len(text) - index - 1}')
else:
    print(f'Sorry, {text} contains non-digits; stopping.')
```

Sure enough, this produces the right output:

```
Enter a number: 2468
2 * 10 ** 3
4 * 10 ** 2
6 * 10 ** 1
8 * 10 ** 0
```

AI Challenge: Plant a Bug

Earlier, you asked AI to add a bug to your program, and you had to find it. This time, you'll add a bug to the program, and challenge AI to find and explain it.

Again, the challenge isn't to see if AI can find the bug. It can. The question is whether you can predict the problem that your bug will cause, and if AI will come to a similar conclusion when it finds the bug. For example, this was my prompt to Gemini:

```
I'm in the 5th chapter of "Python for non-programmers," which
talks
about "for" loops. I have just written a program that asks the
user to
enter a number. If it contains only digits, then I want to print
each
digit with the power of 10 by which it should be multiplied. For
example, if the user enters 2468, then the program should output:
```

```
    2 * 10 ** 3
    4 * 10 ** 2
    6 * 10 ** 1
    8 * 10 ** 0

Here is my program:

    text = input('Enter a number: ').strip()

    if text.isdecimal():
        for index, one_digit in enumerate(text):
            print(f'{one_digit} * 10 ** {len(text) - index}')
    else:
        print(f'Sorry, {text} contains non-digits; stopping.')

However, I have added a bug to my original, working program. Can
you
find it, and tell me what the implications will be if I don't
remove
the bug?
```

Gemini responded:

```
That's a great exercise for learning about for loops and
enumerate!

The bug in your program is in the calculation of the exponent
(the
power of 10).

🐛 The Bug

The line that calculates and prints the expression is:

    print(f'{one_digit} * 10 ** {len(text) - index}')

The bug is here: **{len(text) - index}**

For the input 2468:

    The length of the string (len(text)) is 4.

    The indices (index) start at 0 and go up to 3 (0, 1, 2, 3).

Digit  Index  Your Code's Exponent  Correct Exponent
```

```
2       0       4-0=4                   3
4       1       4-1=3                   2
6       2       4-2=2                   1
8       3       4-3=1                   0
```

❗ Implications

If you don't remove this bug, your program will always print an exponent that is one greater than the correct one, except for the very
last digit, where it will be an exponent of 1 instead of 0.

For the input 2468, your program currently outputs:

```
2 * 10 ** 4
4 * 10 ** 3
6 * 10 ** 2
8 * 10 ** 1
```

This is incorrect because the number 2468 is equal to 2×103+4×102+6×101+8×100.

✅ The Fix

To fix this, you just need to subtract 1 from your current exponent
calculation:

```
    print(f'{one_digit} * 10 ** {len(text) - 1 - index}')
```

Corrected Program:

```
    text = input('Enter a number: ').strip()

    if text.isdecimal():
        for index, one_digit in enumerate(text):
            # FIX: Subtract 1 from the length to get the highest
power (len - 1),
            # then subtract the index.
            print(f'{one_digit} * 10 ** {len(text) - 1 - index}')
    else:
        print(f'Sorry, {text} contains non-digits; stopping.')
```

Would you like to try modifying your program to handle floating-point
numbers (numbers with a decimal point) as well?

Good news! Gemini found the bug and even offered a (correct) solution.

Controlling Our Loop

You've now seen how a `for` loop can be used to iterate over a sequence of values, such as characters in a string, or numbers in a range. But what happens if you don't want to iterate over the entire sequence? What if, after several iterations, you have reached your goal — or by contrast, you realize that you won't reach your goal, and you should stop? Or that the current iteration is a waste of time, and you should move on to the next one?

Python provides two statements that allow you to control your loops:

- `break` stops the loop immediately, continuing execution of the program after the loop body. It is used when the loop has achieved its goal. For example, if you're searching in a string for a particular character, and you found it, there's no reason to continue with the loop. Using `break` lets the computer continue on to more interesting, useful things.
- `continue` stops the current iteration, returning to the `for` line for the next iteration. For example, if the current character is irrelevant or illegal, and you want to go on to the next one, you can use `continue`.

For example, consider the following program:

```
text = 'abcde'
stop_at = 'd'

print('Start')
for one_character in text:
    if one_character == stop_at:
        break

    print(one_character)
print('Done')
```

The preceding code iterates over every character in `text`, printing each of them. But if `one_character` is the same as `stop_at`, then it immediately ends the loop. This doesn't mean that the program ends; execution continues after the loop body, as you can see from the output:

```
Start
a
b
c
Done
```

You can put `break` anywhere you want in a loop body, but it really only makes sense inside an `if` statement, breaking out of the loop only if a particular condition is `True`. If there is more than one reason why you might want to break out of a loop, you can have multiple `break` statements, each inside its own `if` condition.

The related `continue` statement tells Python to stop the current iteration and go on to the next one. For example, if you are iterating over a sequence and have an illegal character, or if you're iterating over a file (as we'll see in Chapter 8) and encounter a blank line, then there's no need to stick around.

As with `break`, it only makes sense to have `continue` inside an `if` statement. For example:

```
text = 'abcde'
to_ignore = 'd'

print('Start')
for one_character in text:
    if one_character == to_ignore:
        continue

    print(one_character)
print('Done')
```

Here is the output from the preceding code:

```
Start
a
```

```
b
c
e
Done
```

Notice that the output includes `e`, unlike the output from the similar program that used `break`. That's because `continue` does not exit the loop, but rather just the current iteration.

Because `continue` just goes on to the next iteration, you will usually want to invoke it at or near the top of the loop body. Putting `continue` toward the end of the loop body doesn't really help much, since Python would normally continue with the next iteration anyway.

Exercise: Sum Digits

In this exercise, you will use `for`, along with `break` and `continue`, to sum the digits that a user enters in a string.

Specifications

1. Define the variable `total` to be 0.
2. Ask the user to enter a string containing digits and nondigits. Assign it to the variable `text`.
3. Go through each character in `text`:
    - If the character is a period (`'.'`), then stop the loop and print `total`.
    - If the character is a nondigit, then tell the user it is being ignored.
    - If the character is a digit, then add its value to `total`, printing the updated `total`.
4. Print `total`.

For example, here is a sample run of the program including both the program's output and the user's input on the first line:

```
Enter text: 12ab3.4
Added 1; total is now 1
Added 2; total is now 3
a is not a digit; ignoring
b is not a digit; ignoring
Added 3; total is now 6
total = 6
```

You can solve this exercise interactively, with the help of an AI tutor.

AI Challenge: Strategy Session

Once again, I want you to have a "strategy session" with an AI chatbot before writing your actual program. That is: tell the program what you're going to do and how you plan to implement it. Ask the chatbot to give you feedback, but not to write any code for you.

AI Challenge: Prompt and Solution

I gave this prompt to ChatGPT:

```
I'm on chapter 5 of an introductory Python book. The exercise I'm
working on says:

1. Define the variable `total` to be 0.
2. Ask the user to enter a string, and assign to the variable
`text`.

3. Go through each character in `text`:

    - If the character is a period (`'.'`), then stop the loop
and
      print `total`.

    - If the character is a nondigit, then tell the user it is
being
      ignored.

    - If the character is a digit, then add its value to `total`,
```

```
      printing the updated `total`.

4. Print `total`.

My plan is as follows:

1. Set `total` to 0.

2. Get the user's input text with `input` and a call to
`str.strip`. Assign the result to `text`.

3. Use a `for` loop to go through each character in `text`. For
each
one, apply the following two tests:

    - Is one_character == '.'? If so, then use `break` to exit
the
      loop.

    - Is one_character.isdecimal()? If so, then invoke int() to
      convert it to an integer, add to total, and print total.

    - If one_character is not a digit, then tell the user we are
      ignoring that character.

4. Print `total`.

Am I missing anything from my strategy here? Please do not write
code.
Just respond to my strategy.
```

Here was its response:

```
Your strategy is solid and very much in the spirit of what the
exercise is asking. You've covered almost everything. There are
just a
few small points to be conscious of:

1. **Order of the checks inside the loop**

   You already described this, but it's important to keep it
explicit:

   1. First: check if the character is `'.'` → if yes, you should
   **print `total` and then break**.
```

2. Else if the character is a digit → convert to `int`, add to `total`, print the updated `total`.

3. Else (everything else) → print a message saying you're ignoring that character.

If you did "non-digit" before you check for `'.'`, then `'.'` would be treated as "ignored" instead of stopping the loop — so your current order (period → digit → otherwise) is correct.

2. **Printing when you see the period**

In your strategy you wrote:

> Is `one_character == '.'`? If so, then use `break` to exit the loop.

The exercise says: "If the character is a period (`'.'`), then stop the loop **and print `total`**."

So the only thing missing from your description is that you should *also* print `total` at that moment, before breaking. (You will also print `total` again after the loop in step 4, which is what the instructions say.)

3. **What happens if there is no period at all?**
With your plan:

* The loop will process all characters (digits update `total`, others are ignored with a message).

* At the end, step 4 prints `total`.

That behavior matches the specification and is perfectly fine. Just be aware that in this case, there will be no "mid-loop" print caused by a period — only the running totals after each digit and the final print at the end.

4. **About stripping the input**

```
   Using `str.strip()` is reasonable: it removes leading/trailing
   whitespace, which you probably don't care about. Any
*internal*
   spaces (like `"3 a 4"`) will still be iterated over and
handled by
   your "non-digit → ignore" rule, which is exactly what you
want.

5. **Non-digit examples**

   Your plan to treat **everything that isn't `'.'` or a digit**
as
   "ignored" — letters, spaces, punctuation, minus signs, etc. —
is
   aligned with the exercise as written. You don't need to treat
`'_'`
   or other characters specially, since the instructions only
mention
   "digit vs. non-digit."

---

So, short answer:

Your strategy is correct and complete *except* for one small
addition
 — when you see the period, you should both **print `total` and
then
break**. Everything else you've planned fits the exercise nicely.
```

As for the solution, the first thing your program needs to do is set up the `total` and `text` variables:

```
total = 0
text = input('Enter text: ').strip()
```

Following that, you need to iterate over every character in `text`, one at a time, using a `for` loop. In that loop, there are three possible conditions:

1. `one_character` is a digit, which means `one_character.isdecimal()` is `True`.

2. `one_character` is *not* a digit, which means `one_character.isdecimal()` is `False`.
3. `one_character` is a `.`, which can be determined with `one_character == '.'`.

Because `'.'` is not a digit, it is important to check for it *before* checking for digits. You can then have an `if` statement that checks whether `one_character.isdecimal()`. If it is a digit, then you can run `int` on it, get an integer value back, and add that to `total`. If it isn't a digit, then you can inform the user:

```
for one_character in text:
    if one_character == '.':
        break

    if one_character.isdecimal():
        total += int(one_character)
        print(f'Added {one_character}; total is now {total}')
    else:
        print(f'{one_character} is not a digit; ignoring')

print(f'total = {total}')
```

Why, in the preceding code, did I use `if`, followed by `if/else`, rather than `if/elif/else`? It's partly a matter of logic, and partly a matter of style.

Remember that multiple `if` statements in a row are evaluated separately. It could be that none of the blocks will run, if all of the conditions return `False`. It could be that all of the blocks will run, if all of the conditions return `True`. And it could be that some will run and some won't.

By contrast, using `if/else`, or `if/elif/else`, guarantees that one, and only one, of the blocks will run. It describes a mututally exclusive set of options.

The thing is, the preceding program could have used `if/elif/else`, because the options are mutually exclusive. But the first `if`, checking if

`one_character == '.'`, feels different and separate from the second two conditions, in no small part because it would lead to an exit from the loop.

The second and third conditions, by contrast, must be in an `if`/`else`, because they are mutually exclusive. Either `one_character` is a digit, in which case its `int` value is added to `total`, or it isn't a digit, in which case the user gets a warning. But it cannot be both.

There isn't anything technically wrong with that code. But I'm a big fan of putting as many `break` and `continue` statements at the top of the loop as possible. It means that the bulk of the loop body will be the default, usual case, which is easier to understand. It also means that the bulk of the loop body can remain unindented, because it won't be in an `if` or `else` block. Here is how you could accomplish that:

```
for one_character in text:
    if one_character == '.':
        break

    if not one_character.isdecimal():
        print(f'{one_character} is not a digit; ignoring')
        continue

    total += int(one_character)
    print(f'Added {one_character}; total is now {total}')

print(f'total = {total}')
```

This loop body starts as before, checking if `one_character` is `'.'`. But then it has another `if` statement, this time using `not` along with `one_character.isdecimal()`. If the character isn't a digit, then the program tells the user and invokes `continue` to go on to the next iteration.

If the program gets through those two `if` statements, then you can be sure `one_character` contains a digit. No more `if` statements are needed; you can invoke `int` on `one_character`, add it to `total`, and go on with the program.

while Loops

`for` loops are extremely common in Python, and you're probably starting to understand why. Most of Python's data structures work well in `for` loops, allowing you to repeat an action for each element of an iterable. You have already seen this for strings and ranges, and as you continue through this book, you'll see even more examples.

There is, however, one problem with `for` loops: they assume that you know how many times you'll want to repeat yourself. Whether you want to do something once for each character in a string, or once for each number in a range, the number of iterations is set from the start. You can exit from a loop early with `break`, but you cannot really have more iterations than the `for` loop was originally meant to run.

In many cases, you don't know how many iterations you'll need. Rather, you know the state that you're looking for, and you want to repeat an action until that state is achieved. For such situations, Python offers a `while` loop.

You can think of a `while` loop as a repeating `if` statement: like an `if`, `while` has a condition. And like an `if`, `while` has a block that is executed if the condition is `True`. The difference, however, is that when the block finishes, Python returns to the `while` statement and its condition. If it is still `True`, then the block executes again.

For example:

```
x = 5

while x > 0:
    print(x)
    x -= 1
```

The preceding code starts by assigning `x` the value `5`. The `while` loop checks if `x > 0`. If so, then the loop body executes, first printing the current value of `x` and then decrementing `x` by 1 — that is, assigning `x` to be 1 less than its current value. When Python reaches the end of the loop body, it returns to the `while` statement, which again checks if `x > 0`.

> **NOTE**
>
> If the loop didn't include the `x -= 1` line, then it would print `x` forever, what we call an "infinite loop." When writing `while` loops, it's important that the condition be based on something that can change, to ensure that the program will exit. In the preceding code, `x` goes down by 1 with each iteration, ensuring that it'll eventually stop.

You can track each iteration of the preceding `while` loop in Table 5-4.

Table 5-4. What happens in `while` loop?

Iteration	`x`	Body runs?	Printed output
Before loop	5	—	—
1st	4	Yes	5 4
2nd	3	Yes	5 4 3
3rd	2	Yes	5 4 3 2
4th	1	Yes	5 4 3 2 1
5th	0	No	5 4 3 2 1
Loop ends	0	No	5 4 3 2 1

Both `break` and `continue` work inside a `while` loop, just as they do inside of a `for` loop:

- `break` exits from the loop entirely, continuing with whatever code exists after the `while` block.

- `continue` ends the current iteration, returning to the `while` statement and its condition.

Many tasks could be expressed as either a `for` loop or a `while` loop. The difference is how you know you want to end: do you want to loop until you reach the end of a sequence? Or do you want to end when you reach a known condition?

I like to make the analogy of a parent who sees a child's clothing on their bedroom floor, and who wants the floor to be cleaned:

- The parent could say, "I want you to pick up each piece of clothing currently on the floor, and put it away." That's similar to a `for` loop, where you give a specific instruction for each item in the sequence.
- Alternatively, the parent could say, "So long as there is at least one piece of clothing on the floor, choose one, pick it up, and put it away." That's similar to a `while` loop, where you state a condition, and repeat the action until the condition is `False`.

Exercise: Sum to 100

In this exercise, you'll ask the user to enter numbers, adding them to a running total. When the total reaches 100 or more, the program stops.

Specifications

1. Set `total` to 0.
2. Repeatedly ask the user to enter a number.
3. If the user enters a nonnumber, then scold them and let them try again.
4. Add the user's entered number to `total`. Print the current `total`.

5. When `total` is 100 or more, then stop asking and print the final `total`.

Example:

```
Enter a number: 30
    total is 30
Enter a number: 20
    total is 50
Enter a number: hello
    hello is not a number; try again
Enter a number: 80
    total is 130
Final total is 130
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The program starts by setting `total` to 0:

```
total = 0
```

The heart of the program is a `while` loop, which checks whether `total < 100` at the top of each iteration. In the first iteration, `total` is 0, so the loop body is guaranteed to execute.

This gives the user a chance to enter a number, which the program receives from the combination of `input` and `str.strip`.

The program checks whether the user's input contains nondigits via `if not text.isdecimal()`. If that condition returns `True`, then at least one character in the user's input is not a digit, and cannot be turned into an integer with `int`. The program prints a warning, and then uses `continue` to move on to the next iteration.

If the program had used `break` instead of `continue`, then a user entering a non-number would have meant exiting from the loop before `total` was

100. Using `continue` allowed the user to recover from this mistake and enter a new, presumably legal, input.

The program then adds `int(text)`, i.e., the integer version of `text`, to `total`, and prints the current total.

At that point, execution returns to the `while` loop, checking whether its condition is still `True`.

How many times will the loop body run? It all depends on what the user enters with each iteration. If the user enters 100 right away, then the loop body will only execute once. But if the user enters 0 with each iteration, then it could potentially last forever.

Here is the full program:

```
total = 0

while total < 100:
    text = input('Enter a number: ').strip()

    if not text.isdecimal():
        print(f'{text} is not a number; try again')
        continue

    total += int(text)
    print(f'total is {total}')

print(f'Final total is {total}')
```

AI Challenge

This time, I want you to give an AI chatbot the specifications that I gave you, and ask it to write a program. Then you should compare your solution with what the AI system wrote.

What similarities do you see? What differences?

Here is what I wrote to Claude:

```
I'm on chapter 5 of an introductory Python book. The exercise I'm
working on says:
```

```
In this exercise, you'll ask the user to enter numbers, adding
them
to a running total. When the total reaches 100 or more, the
program
stops.

. Set total to 0.
. Repeatedly ask the user to enter a number.
. If the user enters a nonnumber, then scold them and let them
try again.
. Add the user's entered number to total. Print the current
total.
. When total is 100 or more, then stop asking and print the final
total.


Please write me a program that'll answer this exercise. I'll then
compare my solution with yours.
```

Claude responded:

```
Here's my solution for the exercise:

```python
total = 0

while total < 100:
    user_input = input("Enter a number: ")

    if not user_input.replace('.', '', 1).replace('-', '',
1).isdecimal():
        print("That's not a number! Try again.")
        continue

    number = float(user_input)
    total += number
    print(f"Current total: {total}")

print(f"Final total: {total}")
```

This solution:
- Starts with `total` at 0
- Uses a `while` loop to keep asking until we reach 100
- Checks if the input is a valid number (handling decimals and
```

```
negatives)
- Scolds the user if they enter something invalid, then lets them
  try again with `continue`
- Adds the valid number to the running total and displays it
- Stops once the total reaches 100 or more and prints the final
result

How does your solution compare?
```

Summary

In this chapter, you saw how loops help avoid violating the "don't repeat yourself" principle in programming. Loops allow you to consolidate logic into smaller, tighter, easier-to-understand sections that make it easier to read, write, debug, and maintain our code. You learned that Python offers both `for` and `while` loops, and that these loops work a bit differently than those in other languages. And you also saw how to exit a loop with `break` or the current iteration with `continue`. You also saw how to accomplish common tasks with Python's loops, whether it's iterating a certain number of times with `range` or getting the index along with each character of a string with `enumerate`.

In the next chapter, you will learn about two new data structures, lists and tuples, both of which are iterable (like strings), but which are more flexible.

Chapter 6. Lists and Tuples

Computer programs deal with a lot of numbers, and also with a lot of text. It's thus no surprise that you have spent the first few chapters of this book learning about integers, floats, and strings. But most programs need to deal with many numbers, such as all of this month's bank transactions, or a patient's temperature over the last week. They might also need to handle many pieces of text, such as the error messages that a program produced over the last week, or all of the email messages in your inbox.

It's certainly possible to use individual variables to handle such individual values. But it's generally easier and more elegant to store values in a *container*, a value that contains other values. You could have all of the documents on your computer in a single, large folder, but you likely put related documents into folders, and even folders within other folders.

That is the thinking behind Python's container types. In this chapter, you'll learn about *lists* and *tuples*, the simplest and most common container types in Python. You'll also see how, in some ways, you can think of a string as a container type — and that Python encourages this kind of thinking. You'll learn about the differences between lists and tuples, how to use each of them, and what sorts of things you can do with them.

It is very common to break a string into a list, and also to combine list elements into a single string. You will see how the `str.split` and `str.join` methods make this possible, and when you would want to do so.

You will also start to understand the difference between *mutable* and *immutable* data structures. So far, all of the values we have used in this book have been immutable, meaning that they cannot be changed. Lists, however, change that, which has a number of implications for your programs.

By the end of this chapter, you will understand when and how to use a list or a tuple, and how to perform some of the most common, useful actions on them.

Lists

A list, as mentioned earlier, is a "container" value. That is, a list isn't really useful by itself; you use a list to hold other things, much like a box or folder. A list can contain any Python values. So a list can hold integers, floats, strings, lists — or any of the other values that you will use in your Python career.

For example, here are three Python lists:

```
numbers = [10, 20, 30, 40]
s_and_p = [6802.99, 6737.93, 6625.84]
names = ['Washington', 'Adams', 'Jefferson']
```

If you put the preceding code into the Python Tutor, it gives the output shown in Figure 6-1.

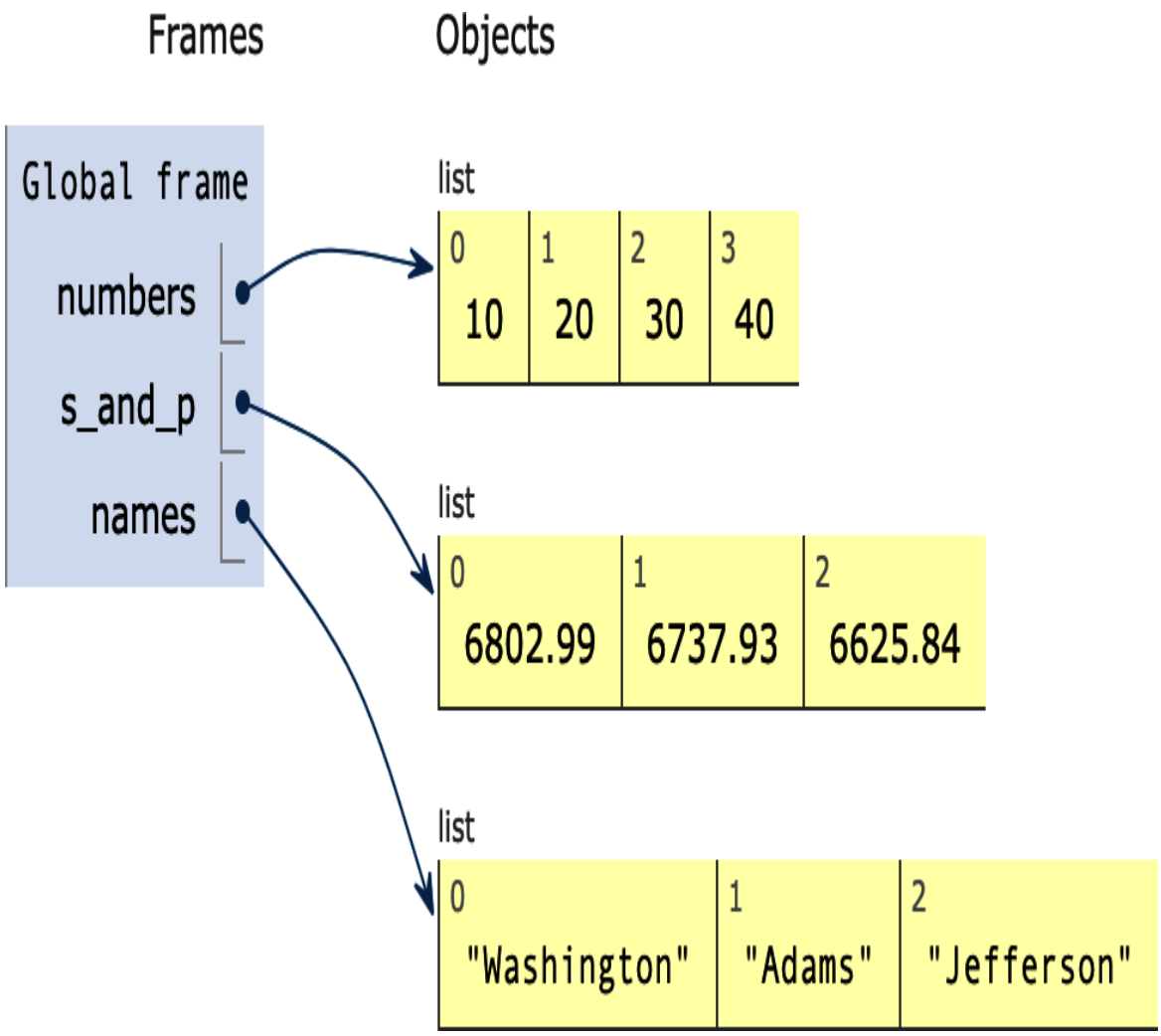


Figure 6-1. Visualization of three simple lists

In theory, a list can contain any combination of different values, such as both strings and floats. However, over time you'll see that it is more common for lists to contain one kind of value, such as strings or integers.

> **NOTE**
>
> The Python Tutor is great for beginners but sometimes uses unfamiliar terminology. For example, Python keeps track of variables in a *frame*. All of the variables you have seen so far are *global*, and that will continue to be the case until you learn about functions in Chapter 9. Python Tutor thus shows all variables in the *global frame*. The values to which variables refer are often known as *objects* in the programming world, which explains that heading above the values.

Defining a List

You define a list with `[]`, and with commas (i.e., `,`) between the elements. From Python's perspective, there isn't any such thing as a "list of strings" or a "list of ints" — there are just lists, and they contain whatever values we want.

As you have seen, Python normally expects the end of a line to mark the end of a statement. However, if you have opened parentheses, then Python will see everything through the closing of the parentheses as a single line. You can thus define a long list across multiple lines, so long as the square brackets are still open:

```
numbers = [10, 20, 30,
    40, 50, 60, 70,
    80, 90, 100]
```

You don't need to put a space after each comma, but it is considered good style to do so. Similarly, it's considered good style not to put a space after the opening `[` or before the closing `]`.

If you have used another programming language before, then this might remind you of what other languages call *arrays*. And yes, there is a fair amount of overlap, both in how they are used and in their behavior. But as you will see later in this chapter, a list's length can change over the course of a program, whereas array lengths are set at creation time and cannot change. Also, arrays can only contain a single type of value; while the convention in Python is to only put one type of value in a list, the language doesn't enforce this. There are actual arrays in Python, but they are not lists, and you won't be using them in this book.

A list without any values is known as the *empty list*, and is written as `[]`. This is similar to the *empty string*, which you write as `''` or `""`. And just as there is no maximum string length in Python, there is also no maximum list length.

NOTE

Who decides what is and isn't good Python style? Changes to the Python language are proposed in "Python Enhancement Proposals," also known as PEPs. Each PEP gets a number, and "PEP 8 – Style Guide for Python Code", is arguably the most famous of them all. It's worth taking a look at PEP 8, which has not changed much over the years, to understand how Python programmers prefer to style their code.

Lists Are Sequences

One of Python's great strengths is its consistency: once you learn how to do something in Python, you will reuse that knowledge numerous times. That's especially true with data structures, which are purposely defined to act similarly, flattening the learning curve somewhat.

This is particularly true for three of the core data structures, described as *sequences* — strings, lists, and tuples. You'll learn more about tuples later in this chapter, but there are numerous similarities between strings and lists. Of course, a string can only contain characters, whereas a list can contain anything, so there are some obvious differences as well.

For example, you already know that you can retrieve an element from a string using `[]` and a numeric index. Lists work exactly the same way:

```
numbers = [10, 20, 30, 40, 50, 60, 70]
print(numbers[2])    # prints 30

i = 3
print(numbers[i])    # prints 40
```

Slices work the same way, too. Remember that the ending index of a slice is *not* included. Also, you can leave out the start or end of a slice, in which case it will go through that end of the list:

```
numbers = [10, 20, 30, 40, 50, 60, 70]
print(numbers[2:5])  # prints [30, 40, 50]
print(numbers[2:])   # prints [30, 40, 50, 60, 70]
print(numbers[:5])   # prints [10, 20, 30, 40, 50]
```

As with a string, you can also search within a list using `in`. But remember that the item for which you're searching, which goes on the *left* side of `in`, must be a precise match for one of the elements in the list:

```
names = ['Washington', 'Adams', 'Jefferson']
print('Washington' in names)   # True
print('Adams ' in names)       # False--notice the space!
print('W' in names)            # False
```

You can also iterate over the elements of a list with `for`. For example, you can use the `len` function to get the length of each of the elements in a list of strings:

```
names = ['Washington', 'Adams', 'Jefferson']

for one_name in names:
    print(f'{one_name}, {len(one_name)} letters')
```

The program outputs:

```
Washington, 10 letters
Adams, 5 letters
Jefferson, 9 letters
```

Exercise: Odds and Evens

In this exercise, you will create a list of integers, summing the odd numbers and even numbers separately.

Specifications

1. Define two variables, `odds` and `evens`, both to be 0.
2. Define a list, `numbers`, containing any integers you want. Make sure that it contains some evens and some odds.
3. Iterate over the list, one number at a time.

    a. If the number is even, add it to `evens`.

    b. If the number is odd, add it to `odds`.

4. Print the values of both `odds` and `evens`.

For example, if `numbers` is `[2, 3, 4, 5]`, then running the program should output:

```
odds: 8
evens: 6
```

Wondering how you can check if a number is odd or even? You can use the `%` operator, known as *modulo*, which returns the remainder after integer division (known as the *modulus*). If a number `% 2` is 1, then it's odd. But if the number `% 2` is 0, it's even. For example:

```
for n in [10, 15]:
    if n % 2 == 0:
        print(f'{n} is even')
    else:
        print(f'{n} is odd')
```

The preceding program prints:

```
10 is even
15 is odd
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

Before doing anything else, you'll need to define the three variables mentioned in the specification:

```
odds = 0
evens = 0

numbers = [2, 3, 4, 5]
```

Next, you'll need to iterate over each element of the list using a `for` loop. In the loop, you'll need to check whether the number is odd or even using the `%` operator. After the loop, you'll then print the values of both `odds` and `evens`. You can even add additional calls to `print` inside the loop, to keep a running total and an update to the person running the code:

```
for one_number in numbers:
    if one_number % 2 == 1:
        odds += one_number
        print(f'{one_number} is odd; odds is now {odds}')
    else:
        evens += one_number
        print(f'{one_number} is even; evens is now {evens}')

print(f'odds = {odds}')
print(f'evens = {evens}')
```

In some ways, this program is similar to the "sum digits" exercise that you did in the previous chapter. However, there are two important differences: first, in "sum digits," you iterated over a string, getting one character at a time. Each character needed to be transformed into an integer before it could be added to the total. While a list could contain strings of digits, you can also just store the integers directly in the list. This removes the need for any translation.

A second difference is that because the list contains integers, and not just digits, they can be as large or as small as you want. You are no longer restricted to numbers between 0 and 9. In this way, lists are truly a container for other values, and there are no restrictions on what those values can be.

If you run the exercise solution in the Python Tutor, you'll get the visual output shown in Figure 6-2.

Print output (drag lower right corner to resize)

```
3 is odd; odds is now 3
4 is even; evens is now 6
5 is odd; odds is now 8
odds = 8
evens = 6
```

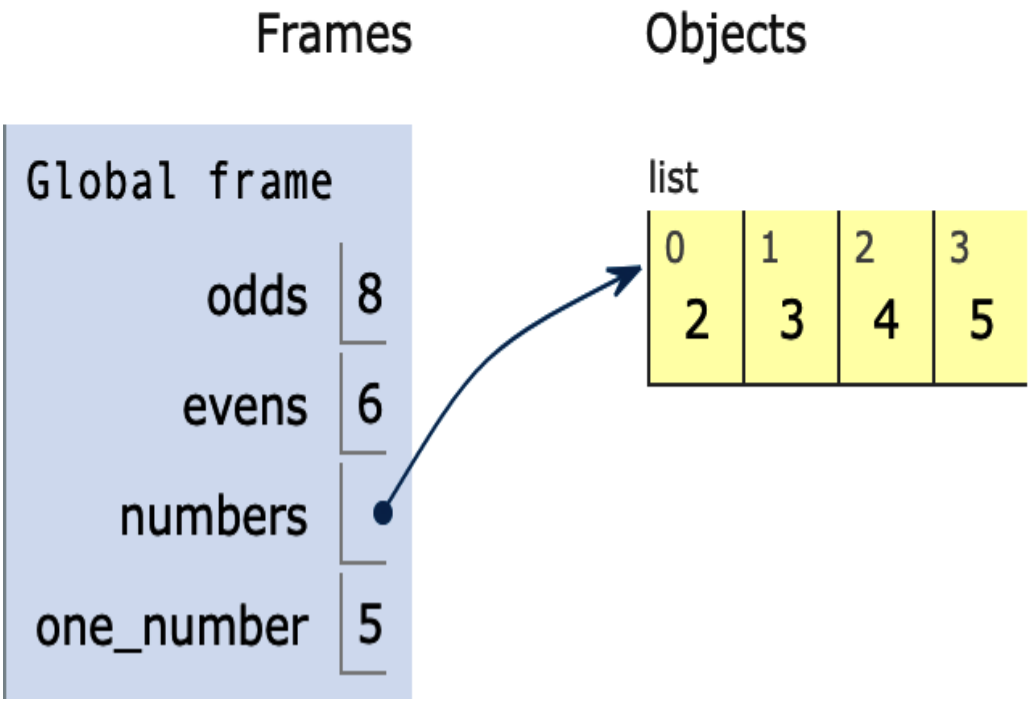


Figure 6-2. Output from the "odds and evens" exercise

Notice that after the program has ended, the `one_number` loop variable contains the final value from the `numbers` list.

AI Challenge: Interview

Once you have written your program and are confident that it is working correctly, enter it into a GenAI chatbot and have it interview you about the code, to solidify your understanding of the program. Here is a transcript of my exchange with Gemini:

I am reading an introductory book on Python, and am currently learning about
lists. I have written this program, which takes a list of integers, identifies
which elements are even and which are odd, and adds each to one of two variables,
`evens` or `odds`:

```python
odds = 0
evens = 0

numbers = [2, 3, 4, 5]

for one_number in numbers:
    if one_number % 2 == 1:
        odds += one_number
        print(f'{one_number} is odd; odds is now {odds}')
    else:
        evens += one_number
        print(f'{one_number} is even; evens is now {evens}')

print(f'odds = {odds}')
print(f'evens = {evens}')
```

Repeatedly ask me questions about my code, interviewing me about what I have written. Then evaluate my answer, and tell me where my explanation is right, and also where it is wrong or could be more precise. Keep asking questions until I ask you to stop.

Gemini responds:

That's a fantastic way to learn! It looks like you've created a simple
program to calculate the sum of odd and even numbers in a list. I'd be
happy to interview you about it.

Let's start with your first question.

? Question 1

You initialize odds = 0 and evens = 0 before the for loop.

What is the purpose of setting these variables to 0 at the very
beginning of your program?

Take your time to explain your reasoning.

I answer:

If I don't set them, then I cannot add to them. Because I'll want to
add integers to them, I need to set both variables to be integers. And
because I want to total numbers, I should start with 0.

Gemini responds:

Your explanation is correct and hits all the essential points.

* "If I don't set them, then I cannot add to them." (Correct): This is
exactly right. In Python, you must define a variable before you can
use it. The act of writing odds = 0 creates the variable name (odds)
and gives it its initial value. If you tried to use odds += one_number
without the initial definition, Python would raise a NameError.

* "Because I'll want to add integers to them, I need to set both
variables to be integers." (Correct/Precise): This is correct. By
assigning 0, you are giving the variables the int data type. Python is
dynamically typed, meaning the type is determined by the
value. Starting with an integer value (like 0) tells Python to

```
treat
these variables as integers, which is necessary for the
subsequent
arithmetic operations (+=).

* "And because I want to total numbers, I should start with 0."
(Correct/Precise): This is the mathematically sound reason. For
an
accumulating sum, the starting point (the additive identity) must
be
zero so that the very first number added is correctly
represented. Any
number plus zero is itself.

Now let's look at the heart of your loop:

    for one_number in numbers:

Describe in your own words what this line of code does.
Specifically,
what is the role of the temporary variable one_number?
```

Lists Are Mutable

In Chapter 4, you saw that once a string is defined, it cannot change:

```
text = 'abcd'
text[2] = '!'     # this results in an error
```

This is because strings are *immutable*. But lists are *mutable*, meaning that they can change:

```
mylist = [10, 20, 30, 40]
mylist[2] = 999

print(mylist)
```

The preceding program prints:

```
[10, 20, 999, 40]
```

Notice that the assignment to `mylist[2]` modified the list's value; the variable `mylist` continues to refer to the same list as before, although that list has changed.

Consider this metaphor: you borrow a T-shirt from your sibling, and then dye it a different color. The shirt has changed, but it still belongs to your sibling. (I am, admittedly, ignoring the effect that this action will have on your relationship!) This is only possible because lists are mutable.

You cannot change a string, but you can create a new string based on an old one, and then assign it back to the original variable. Stretching our analogy somewhat, you could buy a new shirt identical to what your sibling already has, but in a different color, give it to them as a present, and then throw the old one into the garbage. They would still have a shirt, and it would seem (for all intents and purposes) as though the original shirt had changed — but really, it's a completely different item of clothing.

Lists mark the first time that we're encountering a mutable data structure. To which you might ask: who cares? Does it really matter?

The answer is: yes, it matters.

The world changes. And if data structures are supposed to represent the world inside a computer program, then allowing them to change makes it possible to reflect those changes. If your program uses a list to track email messages, employees, or items on a restaurant menu, then it makes sense that the list should change as the inbox, company, and restaurant go through changes.

However, mutable data structures also raise some thorny issues. For example, Python allows two variables to refer to the same value. If they refer to a mutable value, and that mutable value changes, then the change is reflected in *both* of those variables. For example:

```
first_list = [10, 20, 30]
second_list = first_list    # both variables refer to the same
list
```

```
first_list[1] = 999
print(second_list)          # prints [10, 999, 30]
```

Visualizing the preceding code in the Python Tutor makes the behavior clear. Figure 6-3 shows the state after `first_list` and `second_list` are assigned to the same list, and Figure 6-4 shows what happens after `first_list[1] = 999`.

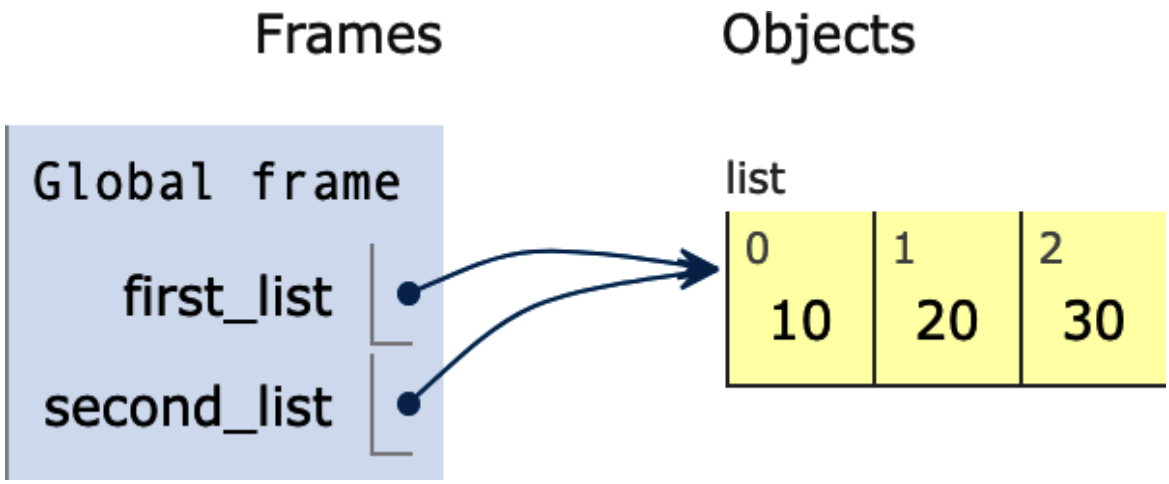
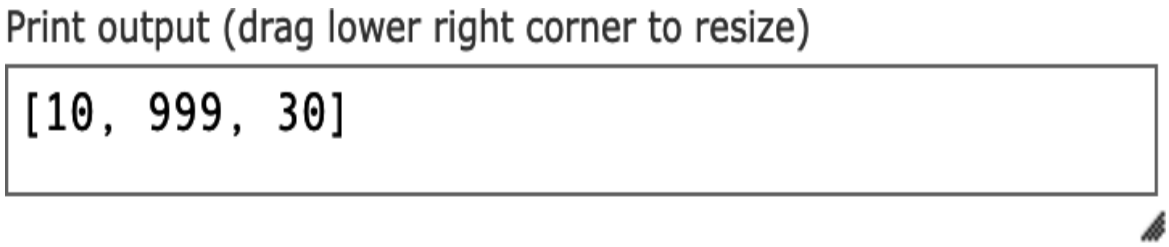


Figure 6-3. Two variables can refer to the same list



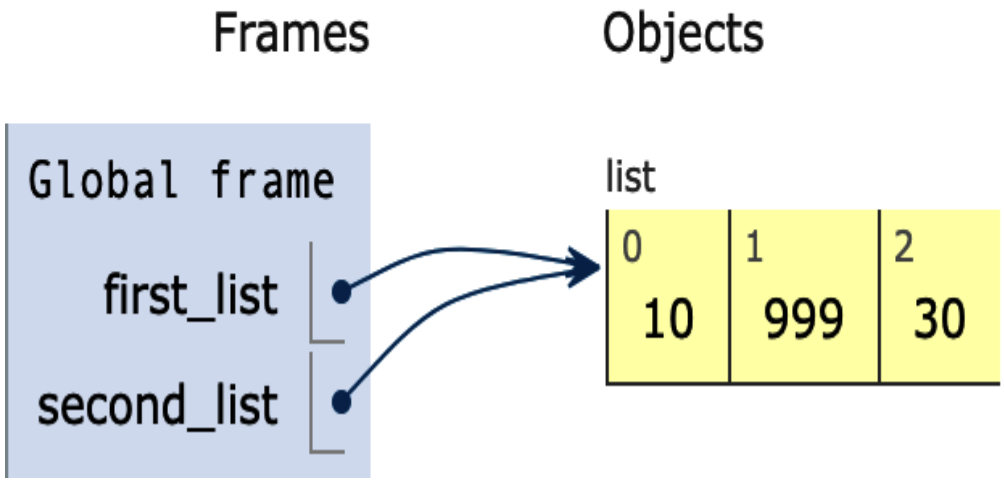


Figure 6-4. Mutating a list affects all of the variables referring to it

This behavior isn't inherently good or bad; it represents a set of trade-offs that every programming language designer makes when creating a language. In the case of Python, having immutable strings boosts the speed and reliability of the language, while making some text-processing tasks harder and more annoying. Mutable lists make it easier to process certain kinds of data, but also introduces the possibility that one part of the program will change a list, and another part will not expect that to happen.

> **NOTE**
>
> The assignment operator = is used for two different things in Python. It can assign a value to a variable, such as `x = [10, 20, 30]`. This creates a value (on the right) and assigns to the variable (on the left). But = can also be used to *mutate* an existing value, as in `x[1] = 888`. In this case, the variable continues to refer to the same list as before. However, the value has changed, even as the variable refers to it.

Three Ways to Mutate

You have already seen that mutable data structures, such as lists, can change. But there are actually three ways in which mutable data structures can change:

- An existing value can be modified (which you've already seen).
- You can add a new value to it, making the data structure longer/larger.
- You can remove an existing value from it, making the data structure shorter/smaller.

If you want to add a new value to a list, you have several options. The most common is the `list.append` method, which modifies the list. This method takes one argument. That value is added to the end of the list, extending its length by 1. For example:

```
mylist = [10, 20, 30]
mylist.append(40)
mylist.append(50)
mylist.append(60)
print(mylist)  # prints [10, 20, 30, 40, 50, 60]
```

Notice that `list.append` always puts a new value at the end of a list. There is a `list.insert` method, which allows you to add a new value at an arbitrary location in the list, but it's less efficient and less common than `list.append`.

> **NOTE**
>
> The `list.append` method modifies an existing list. It does *not* return a new list or a modified list. In fact, it returns a special Python object known as `None`, which is a way for Python to say, "There's nothing to see here." For this reason, you should *never* put a call to `list.append` on the right side of assignment. It's tempting to say `mylist = mylist.append(100)`, but after that code executes, `mylist` will now contain `None` — not the modified list, or any list at all! Just invoke `list.append`, without any = anywhere near it, and you'll be fine.

What if you want to add several values to the end of a list? It's tempting to do the following:

```
mylist = [10, 20, 30]
mylist.append([40, 50, 60])
print(len(mylist))    # prints 4
```

However, `list.append` takes whatever value you give to it and adds it to the end of the list. That's right — the first three elements of `mylist` are integers, but the fourth is a list! The preceding code appended a single list value to `mylist`, rather than three integer values. For that reason, `len(mylist)` returns 4. The visualization in Figure 6-5 from the Python Tutor might make it clearer.

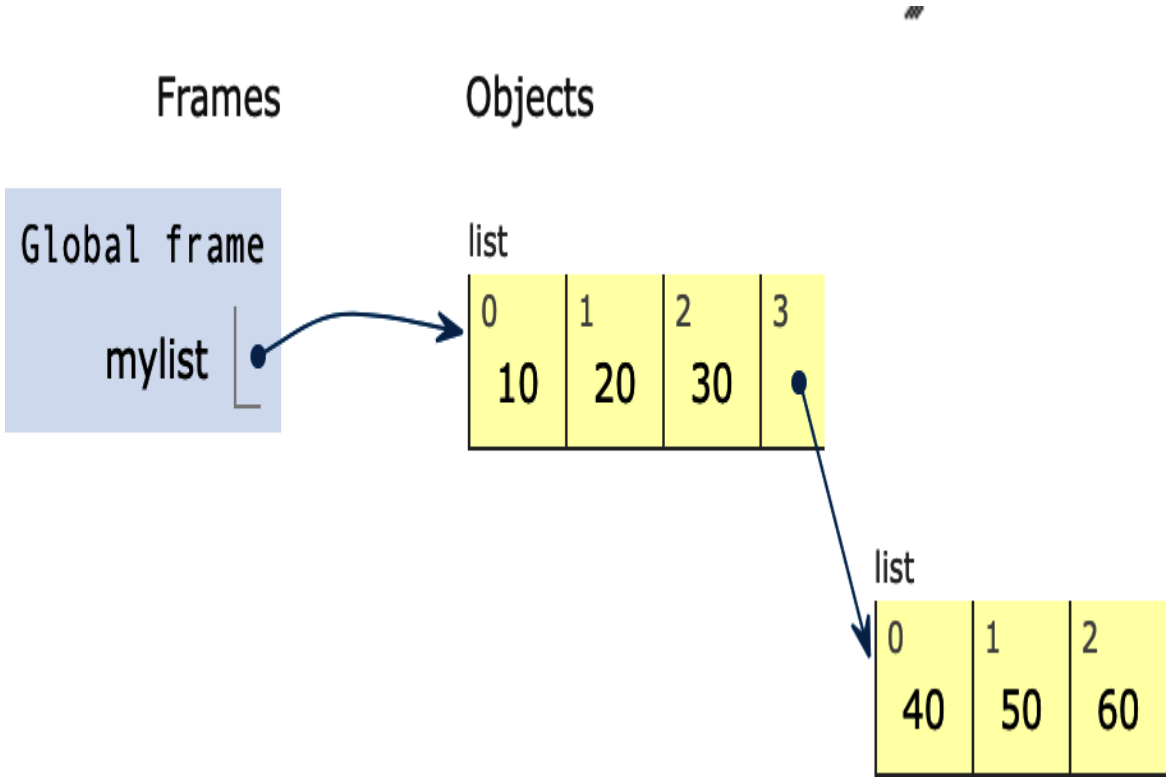


Figure 6-5. `mylist` after appending a list to it

To add multiple values to a list, you should use the `list.extend` method:

```
mylist = [10, 20, 30]
mylist.extend([40, 50, 60])
print(len(mylist))    # prints 6
```

`list.extend` is similar to `list.append`, except that it iterates over its argument, appending each element to the end of the list. The Python Tutor visualization in Figure 6-6 illustrates this.

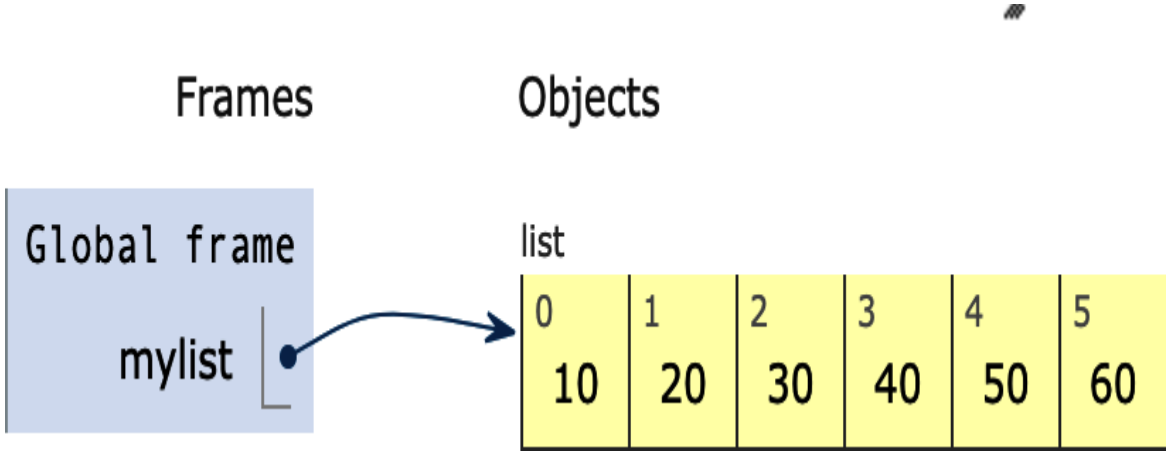


Figure 6-6. `mylist` after running `extend` on it

Finally, you can remove a list element by invoking `list.pop`. By default, `list.pop` removes and returns the final value in the list, the element currently at index -1. However, you can also pass it an integer value, in which case it removes and returns the item at that index:

```
mylist = [10, 20, 30, 40, 50, 60]
print(mylist)  # [10, 20, 30, 40, 50, 60]

mylist.pop()
print(mylist)  # [10, 20, 30, 40, 50]

mylist.pop()
print(mylist)  # [10, 20, 30, 40]

mylist.pop(0)
print(mylist)  # [20, 30, 40]
```

Exercise: Vowels, Digits, and Others (List Edition)

In Chapter 5, you implemented a simple "vowels, digits, and others" counter. You'll now implement a variation on that program, one that not only counts the characters in each category, but stores the characters themselves in lists.

Specifications

Here's what you need to do:

- Define three variables, `vowels`, `digits`, and `others`. Assign the empty list `[]` to each of them.
- Ask the user to enter some text and assign it to the variable `text`.
- Go through the string, one character at a time.
  - Each time you encounter a vowel (a, e, i, o, or u), append the character to `vowels`.
  - Each time you encounter a digit (0-9), append the character to `digits`.
  - In all other cases, append the character to `others`. This includes punctuation and whitespace characters.
- Print the values of `vowels`, `digits`, and `others` on the screen.

Example:

```
Enter text: hello!! 123

vowels: ['e', 'o']
digits: ['1', '2', '3']
others: ['h', 'l', 'l', '!', '!', ' ']
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

We'll start by defining three empty lists, where we'll store the characters we classify, then asking the user to enter some text:

```
vowels = []
digits = []
others = []

text = input('Enter text: ').strip()
```

We can now iterate over the characters in `text`, one at a time — just as we did in the previous version of this program. One minor change since the previous version is that I ran `str.lower` on `text`, such that we'll catch vowels regardless of whether they are capitalized or lowercase.

Another, more significant change, is that the program no longer increments a counter indicating how many characters of each type we found. Instead, it invokes `list.append` on one of the three lists — `vowels`, `digits`, or `others`, adding `one_character` to the end of the list.

The tests, meanwhile, are the same as they were in the previous version of this exercise:

- Check for vowels with `if one_character in 'aeiou'`, using `in` to search in a string.
- Check for `digits with str.isdecimal()`.

If the character is neither a vowel nor a digit, then append it to the "others" list.

```
for one_character in text.lower():
    if one_character in 'aeiou':
        vowels.append(one_character)
    elif one_character.isdecimal():
        digits.append(one_character)
    else:
        others.append(one_character)
```

Finally, the program uses f-strings to display each of the lists, along with their names:

```
print(f'{vowels=}')  # Displays the "vowels=", then the value of
vowels
print(f'{digits=}')
print(f'{others=}')
```

Here's the full solution:

```
vowels = []
digits = []
others = []

text = input('Enter text: ').strip()

for one_character in text.lower():
    if one_character in 'aeiou':
        vowels.append(one_character)
    elif one_character.isdecimal():
        digits.append(one_character)
    else:
        others.append(one_character)

print(f'{vowels=}')
print(f'{digits=}')
print(f'{others=}')
```

AI Challenge: Rubber-Duck Programming

This time, I want you to develop the solution by using the AI chatbot for *rubber-duck programming*. Tell the GenAI what you're planning to do at each step. Any feedback you get is icing on the cake; the real point of rubber-duck programming is that you express your ideas out loud. Doing that forces you to think through what you're doing, and avoid many foolish mistakes.

Here's an example of a rubber-duck session that I had with ChatGPT, implementing the preceding program:

```
I'm reading an intro Python book, and I'm currently on chapter 6,
"Lists and Tuples." The book has asked me to solve the current
exercise using a form of rubber-duck programming. I'll tell you
about
each stage of the program, and what I'm planning to do. Don't
write
code for me, but do indicate where my plans seem good, bad, or
otherwise.

The overall goal of this exercise ("vowels, digits, and others")
is to
get an input string from the user, and then sort each of its
characters into one of three lists--`vowels`, `digits`, and
`others`.
```

```
My plan is to start by assigning each of those variables an empty
list. Then I'll get input from the user:

```python
vowels = []
digits = []
others = []

text = input('Enter your text: ').strip()
```

What do you think of my strategy and code so far?
```

ChatGPT responded:

```
This is a solid start, and your strategy matches the problem
well.

* **Three empty lists:**
✅ Perfect. It sets you up to "classify" characters as you loop.

* **Getting input:**
✅ Good. You're getting a single string to analyze.

A few small notes to think about:

**`.strip()`**:
👍 Fine, but be aware of what it does. It removes whitespace
**only at the
start/end**, not spaces in the middle.

** If the exercise expects spaces/newlines to be treated as
"others,"
stripping will *remove* leading/trailing spaces you might
otherwise classify.
That might be totally OK, just make sure it matches what the
exercise expects.

* **Variable name `text`:**
✅ Clear and descriptive.

Next step (conceptually): you'll want to **loop over each
character**
in `text`, then decide which list it belongs in based on
membership/tests (vowel? digit? otherwise).
```

I then continued:

````
Next, I'll indeed iterate over each character, checking them in
turn:

```python
for one_character in text.lower():
    if one_character in 'aeiou':
        vowels.append(one_character)
    elif one_character.isdecimal():
        digits.append(one_character)
    else:
        others.append(one_character)
```

What do you think of this next step?
````

ChatGPT responded:

````
This is a very good plan, and the logic reads cleanly.

What's strong here:

* **Single pass over the string:**
✅ Efficient and simple.

* **`text.lower()` + `in 'aeiou'`:**
✅ Nice way to make vowel detection case-insensitive without
extra checks.

* **`elif one_character.isdecimal()`:**
✅ Correct for catching `'0'..'9'` (and it also catches some
other Unicode
digit characters, which is usually fine in an intro exercise).

* **Clear "catch-all" else:**
✅ Ensures every character ends up somewhere.

Tiny things to be aware of:

* Because you iterate over `text.lower()`, you're **storing
vowels as lowercase**
  even if the input had uppercase vowels. That's usually fine
(and often
  intended), but if the exercise wanted you to preserve original
````

```
characters,
  you'd instead check a lowercase version *without* losing the
original.

Overall: ✅ your approach is exactly what rubber-duck programming
should sound
like — straightforward, explicit, and easy to debug.
```

Turning Strings into Lists

You've already seen that you can invoke `int` to get an integer based on a string value. Or you can go the other way, invoking `str` to get a string based on an integer value. Generally speaking, if you have a value of one type and want to transform it into another type, you invoke the destination type as a function on the value. For example:

```
int('5')    # returns the int 5
int(5.0)    # returns the int 5
str(5)      # returns the string '5'
float('5')  # returns the float 5.0
```

These conversions seem fairly straightforward, even obvious. It's less obvious how you might transform a string into a list. If you try, you'll get the following result:

```
list('abcd')  # returns the list ['a', 'b', 'c', 'd']
```

Invoking `list` on a string returns a new list of one-character strings, each taken from the string. The returned list will have the same number of characters as the string, just as individual list elements. This is fine, but there are times when this is not the most obvious or useful transformation:

```
list('abc def')  # returns the list ['a', 'b', 'c', ' ', 'd',
'e', 'f']
```

As humans, we look at the string (`'abc def'`) as two words, and might assume that Python would return a list of two strings, each three characters

long. But no, when you invoke `list` on a string, you'll always get a new list of the same length as the string.

There are times, though, when it makes more sense to break a string into pieces, turning each of those pieces into a list element — such as a string containing words, separated by spaces. It might be nice to turn `'hello out there'` into a list of three elements, with each element containing one word, rather than just one character.

This is precisely what Python's `str.split` method is for:

- It's a string method, so you run it on a string.
- It always returns a list of strings.
- The separator used to break the string into elements is passed as an argument to `str.split`.
- Every occurrence of the separator is where Python breaks the string.
- The separator itself is removed when creating the output list.

For example:

```
'abc def'.split(' ')        # returns the list ['abc', 'def']
'abc def'.split('d')        # returns the list ['abc ', 'ef']
'abc:def:ghi'.split(':')    # returns the list ['abc', 'def',
'ghi']
```

It's more common to run `str.split` on a string that has been assigned to a variable:

```
text = 'abc def'
text.split(' ')        # returns the list ['abc', 'def']
text.split('d')        # returns the list ['abc ', 'ef']

text = 'abc:def:ghi'
text.split(':')        # returns the list ['abc', 'def', 'ghi']
```

If the separator string isn't found in the string, then the method returns a one-element list containing the original string:

```
text = 'abc:def:ghi'
text.split('q')        # no 'q', so it returns the list
['abc:def:ghi']
```

As you can imagine, it's particularly common to invoke `str.split` on strings that contain sentences, in which words are separated by spaces. For example:

```
text = 'this is a test'
text.split(' ')   # returns ['this', 'is', 'a', 'test']
```

What happens, though, if the string contains multiple spaces between words? This is quite common when working with text written by actual people:

```
text = 'this  is   a    test'
text.split(' ')   # returns ['this', '', 'is', '', '', 'a', '',
'', '', 'test']
```

This is another case of the computer doing what you told it to do, not what you really wanted it to do: the string contained multiple spaces in a row. But `str.split` doesn't care; it knows that every time it sees the separator character — `' '` in this case — it should end the current string, and start on a new one. Which means that if there are two spaces in a row, you will have an empty string. Three spaces in a row leads to two empty strings.

While you can understand this from a logical perspective, it makes `str.split` far less useful when trying to break text into lists for analysis. Fortunately, this is a common problem — and generally speaking, if you're encountering a common problem in Python, the odds are good that someone before you has already solved it.

In this case, the solution is simple and elegant: Invoke `str.split` without any argument, putting nothing at all inside the `()`. When you do

that, `str.split` treats any whitespace characters — not only space, but also characters such as the newline character `'\n'` — in any number and combination to be a single delimiter. For example:

```
text = 'this  is   a    test'
text.split()   # returns ['this', 'is', 'a', 'test']
```

You can use any string as a separator, including a multicharacter string. For example:

```
text = 'this::is::a::test'
text.split('::')   # returns ['this', 'is', 'a', 'test']
```

However, `str.split` doesn't support multiple different separators. So you cannot say that both `':'` and `'|'` are separators.

Exercise: Counting Capitalized Words

In this exercise, you'll ask the user to enter a sentence. You will then count the number of capitalized words (i.e., words that start with a capital letter) in the sentence, printing that number on the screen.

Specifications

Here's what you need to do:

- Define a variable, `total`, to be 0.
- Ask the user to enter some text, and assign it to the variable `text`. We can assume that the user will only enter words, without any punctuation.
- Go through each word in the sentence, one at a time.
- If the first letter in the word is capitalized, then add 1 to `total`. Ignore capital letters that aren't at the start of a word.

- Print the value of `total`.

Example:

```
Enter a sentence: My name is Reuven
total is 2
```

You can solve this exercise with the help of an AI tutor.

Solution

The program needs to start by setting `total` and `text`:

```
total = 0

text = input('Enter a sentence: ').strip()
```

Next, you need to iterate over each word in the sentence. If you were to run a `for` loop on `text`, then you would get each character in `text`. But you actually want each word in the sentence. To get that, you need to run `text.split()`. That returns a list of strings, the words in the sentence. You can run a `for` loop on those words:

```
for one_word in text.split():
```

But what should be in the loop body? How can you know if a word's first letter is capitalized? There are a number of ways to handle this, but perhaps the simplest is to take the first letter and run `str.upper()` on it, getting back a capital letter. If `str.upper()` on the first letter is equal to the first letter without running `str.upper()`, then the letter must already be capitalized:

```
for one_word in text.split():
    if one_word[0] == one_word[0].upper():
        total += 1

print(f'total = {total}')
```

Here's the full solution:

```
total = 0

text = input('Enter a sentence: ').strip()

for one_word in text.split():
    if one_word[0] == one_word[0].upper():
        total += 1

print(f'total = {total}')
```

AI Challenge: Strategy Session

This time, I want you to feed the specification — i.e., the instructions — that I gave you for this exercise into a GenAI chatbot, and ask it to generate a solution. You should then compare your solution with the one that the AI created. How are they similar? How are they different?

For example, I gave the following to Gemini:

```
I'm reading an intro Python book. The latest exercise, in a
chapter
called, "Loops, lists, and tuples" comes after teaching about
`str.split` and `str.join`. I'm going to give you the
specification
that the book gave for an exercise, and I would like for you to
solve
it, producing Python code. I will then compare my code with
yours.

Here are the instructions:


In this exercise, you'll ask the user to enter a sentence. You
will then
count the number of capitalized words (i.e., words that start
with a
capital letter) in the sentence, printing that number on the
screen.


* Define a variable, `total`, to be 0.
```

```
* Ask the user to enter some text, and assign it to the variable
`text`. We can
  assume that the user will only enter words, without any
punctuation.

* Go through each word in the sentence, one at a time.

* If the first letter in the word is capitalized, then add 1 to
`total`.
  Ignore capital letters that aren't at the start of a word.

* Print the value of `total`.
```

Gemini responded:

```
# 1. Define a variable, total, to be 0.
total = 0

# 2. Ask the user to enter some text, and assign it to the
variable text.
# We can assume that the user will only enter words, without
# any punctuation.

text = input("Please enter a sentence (words only, no
punctuation): ")

# To go through each word, we first need to split the sentence
into a
# list of words. By default, str.split() splits the string by
whitespace.

words = text.split()

# 3. Go through each word in the sentence, one at a time.

for word in words:

    # 4. If the first letter in the word is capitalized, then add
1 to
    # total.  We can check if the first letter is capitalized
using
    # str.isupper() on the first character.  We must also ensure
the
    # word is not empty before accessing the first character
(though
    # split() usually handles this).
```

```
        if word and word[0].isupper():
            total += 1

    # 5. Print the value of total.

    print(f"The number of capitalized words in the sentence is:
    {total}")
```

As you can see, there are some similarities between the versions. But there are some differences, too. Understanding these differences can help you to better understand how Python works, and which techniques are considered most conventional.

Turning Lists Back into Strings

In the previous section, you saw that `str.split` is a great way to get a list based on the contents of a string. But what about going in the other direction? What if you want to take a list of strings and turn it back into a string?

If all you want is a string based on a list, you can invoke `str`. However, that's unlikely to give you the results that you wanted:

```
mylist = ['this', 'is', 'a', 'test']
text = str(mylist)

print(text)    # prints the string "['this', 'is', 'a', 'test']"
```

In other words, turning a list into a string gives you a string that starts with `'['`, ends with `']'`, and includes each of the elements, along with the commas separating them, which is good for displaying a list on the screen, but isn't quite the opposite of `str.split` that you might be looking for.

For that, we'll need to use the `str.join` method. But as you can see from its name, `str.join` runs on a *string*, not on a list. This is a bit surprising to many people, including people coming to Python from other programming languages that have similar methods. Here's the basic idea:

- You'll need a string, which I call the "glue," that will go between the list elements.
- Invoke `str.join` on the glue string.
- Pass the list of strings as an argument to `str.join`.
- The method returns a new string, with the glue put between the list's elements.

For example:

```
mylist = ['this', 'is', 'a', 'test']

output = ' '.join(mylist)
print(output)    # prints 'this is a test'
```

You can use any string you want as the glue:

```
mylist = ['this', 'is', 'a', 'test']

print(''.join(mylist))     # prints 'thisisatest'
print(', '.join(mylist))   # prints 'this, is, a, test'
print('**'.join(mylist))   # prints 'this**is**a**test'
print('\n'.join(mylist))   # prints each word on its own line
```

As you can see, the glue string can contain any number of characters, including zero characters — in which case the list's elements are squished together without any separator between them.

Exercise: Capitalized Words (Part 2)

In the previous exercise, you counted the number of capitalized words that the user entered. In this exercise, you'll collect the capitalized words themselves, printing them for the user.

Specifications

- Define a variable, `output`, to be an empty list, `[]`.
- Ask the user to enter some text, and assign it to the variable `text`. We can assume that the user will only enter words, without any punctuation.
- Go through each word in the sentence, one at a time.
- If the first letter in the word is capitalized, then append it to `output`. Ignore capital letters that aren't at the start of a word.
- Using `output`, print all of the capitalized words in the sentence, separated by commas.

Example:

```
Enter a sentence: My name is Reuven
My, Reuven
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

This exercise required you to change the previous version of this program in three places:

1. Initially set `output` to be `[]`, the empty list, replacing the `total` variable, which had been set to 0.

2. Instead of `total += 1` each time a new capitalized word was found, you needed to invoke `output.append(one_word)`, adding `one_word` to the end of the `output` list.
3. Instead of printing the value of `total`, you needed to invoke `str.join`, using the string `', '` (comma + space).

The resulting code is:

```
output = []

text = input('Enter a sentence: ').strip()

for one_word in text.split():
    if one_word[0] == one_word[0].upper():
        output.append(one_word)

print(', '.join(output))
```

Notice that the program never assigns the result of `str.join` to a variable. The resulting string is just passed as an argument to `print`.

GenAI Task: Find the Bug

Once you have finished writing the program, ask an AI chatbot to introduce a bug, which you then have to find. See what kinds of bugs can be introduced with only small variations in the code!

Here's what I wrote to Claude:

```
I'm reading an intro Python book, and just finished an exercise
in which I had
to track (and then print) the words starting with a capital
letter. Here is the
code I wrote, which works just fine:

output = []

text = input('Enter a sentence: ').strip()

for one_word in text.split():
```

```
    if one_word[0] == one_word[0].upper():
        output.append(one_word)

print(', '.join(output))
```

I want you to introduce a bug into this code, and see if I can find and
explain it. Respond to my explanation.

Here was Claude's response:

Great exercise idea! Here's your code with a bug introduced:

```python
output = []
text = input('Enter a sentence: ').strip()
for one_word in text.split():
    if one_word == one_word.upper():
        output.append(one_word)
print(', '.join(output))
```

Try running this with a few different inputs and see what happens. Once you've identified the bug, explain:

1. What's different from your original code
2. What behavior you're seeing
3. Why it's happening

Let me know what you find!

Here was my response:

Ah, you're running str.upper on one_word, rather than one_word[0]. Which means
that we'll only keep the word in output if all of the letters are capitalized.
That's not what we want, though; we want to check that just the first letter is
capitalized.

Here was its response:

```
Excellent diagnosis! You've got it exactly right.

The bug is that `one_word.upper()` transforms **all** letters in
the word to
uppercase, not just the first one. So:

- `"Hello"` would be rejected (because 'e', 'l', 'l', 'o' are
- lowercase) `"HELLO"` would be accepted (all uppercase)
`"HeLLo"`
- would be rejected (mixed case)

Your original code with `one_word[0].upper()` correctly checks
only the first
character, which is what you want for detecting words that start
with a capital
letter.

This is a common mistake when working with string methods -
mixing up
`.upper()` (checks the whole string) with checking individual
characters. The same issue can happen with `.lower()`, etc.

Nice work spotting it! Ready for another bug to find, or would
you
like to move on?
```

Tuples

So far, you have learned about two members of the *sequence* family in Python, namely strings and lists. That family of data structures has a third member, *tuples*, which seem a bit weird and even unnecessary when you first learn about them. But they are a core data structure in Python, and they do show up from time to time even if you don't explicitly use them — so you do have to know something about tuples.

First, compare the two sequence types you've already used:

- Strings contain individual characters, and are immutable.
- Lists contain anything at all, and are mutable.

Tuples are a combination of these two attributes: like lists, they can contain any values at all. But like strings, they are immutable.

> **NOTE**
>
> You can pronounce "tuple" as either "TOO-pull" or "TUH-pull." I tend to use the first pronunciation, but you'll find people in the Python world in both camps.

Nearly everyone who learns about tuples asks several questions, starting with: so tuples are basically immutable lists? Why do we need them, anyway? Other programming languages do just fine without tuples.

It's true, many other high-level languages do just fine with the equivalent of Python's lists, and don't have anything corresponding to tuples. Moreover, from a purely technical perspective, tuples are indeed just immutable lists.

In practice, however, tuples are used differently from lists: whereas the convention in Python is for lists to contain values of a single type, the same convention encourages us to put more than one type in a tuple. That's because tuples are supposed to represent database records, where each field is of a different type. For example, if you want to store someone's name (a string) and shoe size (an integer), you would likely use a tuple, rather than a list.

Python won't complain if you create a tuple whose elements are all of one type. Nor will it say anything if a list contains elements of different types. These are conventions among coders. But they're pretty deeply set in the Python community, and aren't worth fighting.

Even if you don't use tuples very much in your day-to-day work, Python uses them quite a bit behind the scenes. When you invoke a function, the arguments to that function are passed via a tuple. When you define a function (as you'll see in Chapters 9 and 10), Python stores the names of the local variables, as well as any default argument values, in a tuple. So tuples are there, even if you don't use them yourself.

You might also see tuples when you connect to other programs and systems. For example, if you read a number of records from a database, they will likely come as a list of tuples. Each tuple will represent a single record in the system; because a record will likely contain different types of values,

Python uses a tuple. But each tuple will contain the same types, and thus the sequence of tuples will be returned as a list. Indeed, lists of tuples are a very common data structure in Python.

Tuple Basics

With that background, you're now ready to work with tuples. Whereas lists are created with square brackets, `[]`, tuples are created with round parentheses, `()`. For example:

```
t = (10, 20, 30, 40, 50, 60, 70)
```

The tuple `t` contains seven integer values. Because tuples are sequences, you can use many of the same techniques that worked with strings and lists. You can retrieve from a tuple with `[]`, using an integer index. Or you can get a slice of values from a tuple, using the same `[begin:end]` syntax that worked with strings and lists. You can search in a tuple using `in`. And you can iterate over a tuple using a `for` loop.

```
print(t[0])     # prints the int 10
print(t[2:5])   # prints the slice (30, 40, 50)

for one_item in t:
    print(one_item)   # prints each number
```

Python uses round parentheses in a number of places, not just in defining tuples. For example, you invoke a function with `()`. You can also set the priority of parts of a math expression. As a result, while you can define an empty tuple with `t = ()`, you cannot define a one-element tuple with `t = (10)`. This code will be interpreted the same as `t = 10`, assigning the integer `10` to `t`. This is a rather surprising part of Python's syntax — but it stems from Python's inability to distinguish between `t = (4 + 6)`, with a math expression inside the `()`, and `t = (10)`.

To resolve this ambiguity, Python requires that one-element tuples have a comma following that one item. As a result:

```
t = ()          # empty tuple, 0 elements
t = (10,)       # tuple with 1 element
t = (10, 20)    # tuple with 2 elements
```

You might also be surprised to find that you don't even need to use parentheses when defining a tuple. It's enough to have commas between the elements:

```
t = 10, 20, 30  # tuple with 3 elements
```

No matter how you define a tuple, it's still immutable, and thus impossible to change:

```
t = (10, 20, 30)
t[0] = '!'
```

This results in the following error:

```
Traceback (most recent call last):
  File "", line 2, in
    t[0] = '!'
    ~^^^
TypeError: 'tuple' object does not support item assignment
```

Once you have defined a tuple, you cannot change its elements or its length.

Tuple Unpacking

Even if you won't be defining many tuples directly, you might still end up using them in one specific way, namely *tuple unpacking*. You already know that you can assign a list to a variable:

```
mylist = [10, 20, 30]
x = mylist
```

After the preceding code executes, both `mylist` and `x` refer to the same list. But Python also allows for the following:

```
mylist = [10, 20, 30]
x, y, z = mylist      # 3 variables and 3 list elements
```

Python looks at `mylist`, and sees that it has three elements. It looks at the tuple of variables on the left side of the = operator, and sees three of them. It then assigns each of the list elements to a separate variable:

```
print(x)    # prints 10
print(y)    # prints 20
print(z)    # prints 30
```

In Chapter 5, you learned about *iterables*. Whenever Python sees an iterable value on the right side of assignment, and multiple variables on the left side of assignment, it assigns each of the elements to a separate variable. The numbers must match up; if there are too many or too few values for the variables, then you will get an error. Note that the right side can contain any iterable, meaning any value that knows how to behave in a `for` loop.

Tuple unpacking is particularly powerful and useful when breaking apart complex data structures into individual variables, which can make the code easier to read, write, and maintain. For example:

```
person = ('Reuven', 'Lerner', 46)
first_name, last_name, shoe_size = person

print(f'Hello, {first_name} {last_name}.')
print(f'I hear you wear size {shoe_size} shoes.')
```

You've already seen an example of tuple unpacking in action in Chapter 5, but you weren't aware of it. In that chapter, you learned about `enumerate`, which gives you indexes along with the elements of an iterable. For example:

```
text = 'abcd'

for index, one_character in enumerate(text):
    print(f'{index}: {one_character}')
```

The preceding code prints each letter along with its index, as you might expect:

```
0: a
1: b
2: c
3: d
```

When you learned about `enumerate`, you also learned that you need to invoke the `for` loop with two variables. But…what if you were to use only one variable? Would the program still run? What would the loop variable contain in each iteration?

```
text = 'abcd'

for one_item in enumerate(text):
    print(one_item)
```

Here is what the preceding program prints on the screen:

```
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
```

As you can see, each iteration over `enumerate(text)` returns a two-element tuple. The first element is the index, an integer. And the second element is a character, the current item over which Python would normally have iterated.

You could then access the index as `one_item[0]` and the value as `one_item[1]`, but that's a bit clunky. It would be better to use tuple unpacking:

```
text = 'abcd'

for one_item in enumerate(text):
    index, one_character = one_item
    print(f'{index}: {one_character}')
```

Of course, this only works because you know that `one_item` is guaranteed to be a two-element tuple. It makes the code a bit more readable, but produces the same output:

```
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
```

But as you've already seen, you can combine those first two lines of the `for` loop into a single one that both iterates and unpacks:

```
text = 'abcd'

for index, one_character in enumerate(text):
    print(f'{index}: {one_character}')
```

Summary

This chapter introduced two new data structures, lists and tuples, that are frequently used in Python programs, both by themselves and together. Both are sequences, but lists are typically used for sequences of the same type, while tuples are used for sequences of different types. The fact that lists are mutable and tuples are immutable doesn't directly affect which of them you should use.

You also saw how you can split a string into a list of strings, and then join a list of strings back into a string, using the `str.split` and `str.join` methods, respectively.

In the next chapter, you'll learn about dictionaries, another important Python data structure. Indeed, you could argue that dictionaries are the most important data structure in Python, because of their flexibility and efficiency. The lessons you've learned with lists and tuples will serve you well as you explore that data structure before moving on to files and functions.

Chapter 7. Dictionaries

The data structures you've learned about so far — numbers, strings, lists, and tuples — might seem fairly simple, but they can be used to store and analyze data in a wide variety of ways. That's particularly true once you learn to combine them, taking advantage of the fact that lists and tuples can be nested infinitely.

For example, if you run an online streaming service, you might use a list of tuples to keep track of the movies in your database. Each tuple would represent one film, and would contain the following fields:

- Name
- Director's name
- Year
- Length
- A list of the top-billed starring actors

If one of your customers wants to watch a movie, how can you tell them whether it's available? The simple answer is a `for` loop: iterate through the list of movies, compare the title (or other details) with each tuple, and when you find a match, answer the user.

There's nothing technically wrong with what I have described. However, it suffers from a problem that is common when working with data, namely that searching can be very slow. In this particular case, the amount of time it takes to search for a movie depends on how many you have in your list of tuples. With only 10 movies in stock, searching will be instantaneous. With 10 million movies in stock, the search will take significantly longer.

Fortunately, computer scientists have thought about this problem for many years. Decades ago, they designed a data structure in which searching is

nearly instantaneous, regardless of how many items you are storing. This data structure, known in Python as a *dictionary* (or *dict*), is perhaps the most important data structure in the language. It is not only elegant, but also quite efficient. Indeed, dictionaries are so efficient that Python itself uses dicts to implement many of its internal needs.

In this chapter, you'll learn how to create, manipulate, modify, and iterate over dicts. You'll learn about several paradigms for using them in your programs, and even how they work behind the scenes. By the end of this chapter, you will feel comfortable using dicts — and might even, like so many Python coders, see them as your primary tool in organizing and working with data.

Intro to Dicts

Dictionaries aren't unique to Python. In other programming languages, they tend to have other names, though:

- Hash tables
- Hashes
- Hash maps
- Maps
- Associative arrays
- Key-value stores
- Name-value stores

You've already seen that strings, lists, and tuples have *elements*, and that each element has a numeric *index* starting with 0. When discussing a dictionary, it's customary to talk about *pairs* or *items*, rather than individual elements. (I prefer to talk about "pairs," since it emphasizes the fact that each item has two parts, but most of the Python world uses "items.") Each pair contains a *key* (rather than an "index") and a *value*.

You've already seen that list and tuple elements can be absolutely anything. The same is true for the values in a dict. But a dict's real strength is that you can also determine the key. You don't need to use integers such as 0, 1, and 2; you can use nearly anything you want. This opens the door to keys that are more expressive and interesting than just integers, such as:

- Employee ID numbers
- Email addresses
- Unique product numbers
- Birthdates

Here are some basic rules for dicts:

- Every key has a value, and every value has a key.
- There are no restrictions on values. They can be of any type, and can repeat.
- Keys must be immutable — which normally means that they will be numbers or strings.
- Within a given dict, the keys must be unique.

> **NOTE**
>
> Technically speaking, dict keys must be *hashable*, rather than immutable. I'll describe what *hashable* means later in this chapter. But for most purposes, especially when starting with Python, the distinction between the two is not worth mentioning.

Defining a Dict

Dictionaries are defined using curly braces. The empty dict, containing 0 key-value pairs, is written as `{}`, and is similar to the empty string (`''`), the empty list (`[]`), and the empty tuple (`()`).

To define a nonempty dict, put the pairs inside the `{}`, with a comma between each pair. Between each key and value, put a colon. For example:

```
d1 = {'a': 10}                        # dict with one pair
d2 = {'a': 10, 'b': 20, 'c': 30}      # dict with 3 pairs
d3 = {1: 'Jan', 2: 'Feb', 3: 'Mar'} # dict with 3 pairs
```

You can use the `len` function to find out how many pairs are in a dict. It's important to remember that when working with a dict, you always have to think about the "elements" as key-value pairs. Thinking about the keys and values separately will make things more confusing, not less so:

```
print(len(d1))     # 1 pair
print(len(d2))     # 3 pairs
print(len(d3))     # again, 3 pairs
```

Just as you can retrieve an element from a string, list, or tuple by using `[]` with the index, you can retrieve the value from a dict by putting the key inside of `[]`:

```
print(d1['a'])     # prints 10
print(d2['b'])     # prints 20
print(d3[1])       # prints 'Jan'
print(d3[3])       # prints 'Mar'
```

Slices don't work when retrieving from a dict. You can only retrieve one value, from one key. If the key doesn't exist, you'll get a `KeyError` exception:

```
print(d1['x'])     # results in a KeyError exception, stopping the
program
```

You can use `in` to search for a key (not a value!) in the dict:

```
print('a' in d1)    # prints True; 'a' is a key in d1
print('x' in d1)    # prints False; 'x' is not a key in d1
print('1' in d3)    # prints False; '1' is not a key in d3
print(1 in d3)      # prints True; 1 is a value (not a key) in d3
```

Notice the two final lines in the preceding example: because the integer `1` isn't the same as the string `'1'`, using the `in` operator on a dict whose keys are integers, and searching for a string, will return `False`.

You can, of course, use a variable instead of a value, both when searching in a dict and when retrieving from it:

```
k = 'a'
print(k in d1)      # prints True
print(d1[k])        # prints 10

k = 'x'
print(k in d1)      # prints False
print(d1[k])        # gives an error
```

> **NOTE**
>
> When using `in` to search in a dict's keys, Python will only return `True` for an exact match. Any difference in type, whitespace, or capitalization will result in a `False` response.

Part of the beauty of a dict is that the keys aren't arbitrary integers, but tie into the actual data you want to store (or retrieve), and that the user might enter. For example:

```
months = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5,
'Jun': 6,
          'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11,
'Dec': 12}

while True:
    text = input('Enter month: ').strip()[:3]

    if text == '':
        break

    if text in months:
        month_number = months[text]
        print(f'Month {text} is number {month_number}')
    else:
        print(f'There is no such month {text}!')
```

This example starts by defining a dictionary, `months`, in which the keys are strings (month abbreviations) and the values are integers (the number of each month). The user is repeatedly asked to enter the name of a month; whatever they enter runs through `str.strip` (to remove leading/trailing whitespace) and then the slice `[:3]`, keeping only the first three input characters.

If the user's text is an empty string, then the loop exits with `break`. Otherwise, the program looks for the user's input text in the `months` dictionary's keys. If it is there, then the user is told the month number for that abbreviation. If it is not there, then the user is scolded appropriately:

```
Enter month: Jan
Month Jan is number 1
Enter month: Feb
Month Feb is number 2
Enter month: July
Month Jul is number 7
Enter month: Heshvan
There is no such month Hes!
Enter month:
```

This program could have been implemented using a list of tuples. But dictionaries make it cleaner and easier to write and understand. And while it's not yet obvious, searching for the month name in the dict's keys is more efficient than iterating over a loop.

A visualization of the `months` dict from the Python Tutor appears in Figure 7-1.

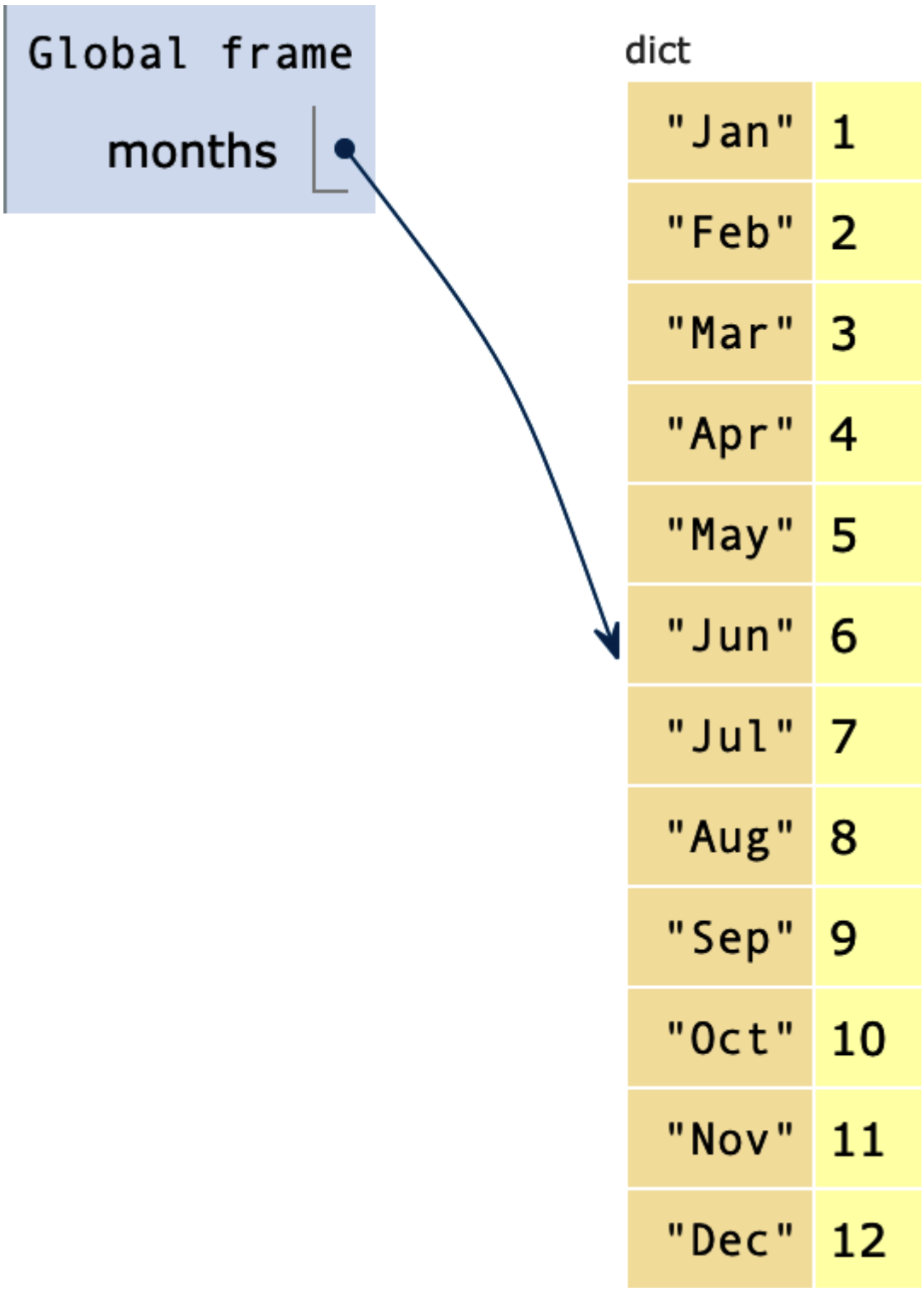


Figure 7-1. Visualization of the `months` dict

It's often easiest to think of a dict as a two-column table. Each row in the table represents one key-value pair. You can use the key to retrieve its

associated value, because the keys are unique. But the values can repeat themselves, so using a value to retrieve a key isn't possible, at least not directly.

Exercise: Pizza Toppings

In this exercise, we'll assume that a plain pizza costs $10. We'll let the user choose one or more toppings for their pizza. Each topping will have a different price. When the user has finished choosing all of their toppings, print the total price of the pizza.

Specifications

Here is how you should solve the problem:

1. Define a dictionary, `toppings`, in which the keys are strings (the names of available toppings) and the values are integers (the price for each topping).
2. Define a variable, `total`, which starts with the base price of the pizza — $10, in our example.
3. Define an empty list, `added_toppings`, which is where we'll keep track of what toppings the user wants.
4. Ask the user, repeatedly, what topping they want to add to their order.
    a. If the user enters an empty string (i.e., just presses `ENTER`), then stop asking and print both the total cost and the toppings they ordered.
    b. If the user enters the name of a key in `toppings`, then print the price of adding that topping, add it to `added_toppings`, and print both `total` and `added_toppings`.

c. If the user enters the name of a topping that is *not* in `toppings`, then scold the user and let them try again.

d. If the user has chosen three toppings, then stop them from adding more, and exit from the loop.

Here is one possible dialogue from someone who successfully adds onion and garlic as toppings, but sadly isn't able to add an elephant, as well:

```
Enter topping:  onion
onion costs 1; total is now 11
Current toppings: onion

Enter topping:  garlic
garlic costs 1; total is now 12
Current toppings: onion, garlic

Enter topping:  elephant
Sorry, but elephant is not a valid topping. Try again.

Enter topping:

Total = 12
Toppings = onion, garlic
```

You can solve this exercise interactively, with the help of an AI tutor.

AI Challenge: Strategy Session

For this exercise, I want you to do a strategy session, running your plans by a GenAI chatbot before actually writing code. There are enough different data structures and logical decisions that you should have a clear plan for what you're going to do before doing it. Once you have fleshed out your thoughts, then you can go ahead and write the code.

AI Challenge: Prompt and Solution

I wrote to Claude:

```
I'm going through an introductory Python book. The current
chapter is
teaching about dicts, and has this exercise:

[The exercise goes here]

I want to strategize with you about how to write this program. I
don't
want you to write code, but rather tell me what you think as I,
step
by step, go through how I want to solve things.

First, I'll define three variables--toppings, total, and
added_toppings. Here's how I'm thinking of defining them:

toppings = {'onion': 1, 'garlic': 1, 'extra cheese': 3,
            'eggplant': 2, 'mushrooms': 2}
total = 10
added_toppings = []

Once I do that, I'll use a while loop to get toppings from the
user. But it
won't be while True, because I don't want an infinite loop. I'll
check while
len(added_toppings) < 3. At the same time, inside of the while
loop,
I'll get the user's order--and if they give me an empty string,
I'll use
break to exit from the loop.

If it's not empty, I'll check whether the string is a key in
toppings.
If yes, then I'll add the price to total and also add the topping
to
added_toppings, the list. If not, then I'll tell the user.

Finally, after exiting the loop, I'll print both total and the
result
of invoking str.join on added_toppings.

What do you think of this strategy?
```

Claude responded:

```
Your strategy is solid and well thought out! Let me walk through
what
```

I like and a few things to consider:

What works well:

Your variable initialization is perfect - the dict structure for toppings with prices as values, starting total at 10, and an empty
list for tracking choices all make sense.

The while len(added_toppings) < 3 condition is clever - it automatically stops at three toppings without needing additional logic. Combined with the break for empty string input, you've got both
exit conditions covered.

Your flow inside the loop is exactly right: get input, check if empty
(break if so), validate against the dict keys, then either add to total/list or scold the user.

Using str.join on added_toppings at the end will give you that nice
comma-separated output.

A couple of things to think about:

When you check if the topping is valid and add its price to total,
you'll need to look up that price from your toppings dict - sounds
like you've got that in mind.

The sample output shows "Total is 11" after each topping, but only
shows "Total = 12" and "Toppings = onion, garlic" at the very end. Do
you want to print the running total after each successful addition
(inside the loop), or just the final total after the loop ends? The
sample dialogue suggests both, actually.

One edge case: what happens if the user immediately presses ENTER without adding any toppings? Your code will handle it fine, but the
output might look a bit odd with "Toppings = " (empty string from join). Not wrong, just something to be aware of.

```
How does this align with what you were planning?
```

The main data structure needed for this program to work is `toppings`, a dict in which the available toppings are the keys, and the price for each topping is an integer. With this structure in place, you can check whether a topping is available by searching the dict using `in`, or retrieve its price using `[]`. You then have to define `total`, starting with the base price of the pizza, and `added_toppings`, an empty list onto which you will add the user's chosen toppings:

```
toppings = {'onion':1, 'garlic':1, 'extra cheese':3,
            'eggplant':2, 'mushrooms':2}
total = 10
added_toppings = []
```

With this in place, it is time to ask the user to choose toppings. But how can you ask the user to enter multiple toppings? A loop is in order. Consider the different possible loops you could use:

- A `for` loop would only make sense if you know how many toppings the user will order. But while the maximum is three, the user might choose fewer than that. (They might even choose not to have any toppings at all, a chilling prospect.)
- A `while True` loop will let the user choose any number of toppings, invoking `break` inside the loop body if the user enters an empty string or if they have already reached three toppings. This will work, but seems overly complex.
- A `while` loop that keeps going, but only so long as there are fewer than three elements in the `added_toppings` list, will also work and will result in cleaner code. But you don't want to check `while added_toppings < 3`, comparing a list with an integer. Rather, you want to compare `while len(added_toppings) < 3`, checking that the number of toppings is less than three.

The `while` loop will thus look like this:

```
while len(added_toppings) < 3:
    order = input('Enter topping: ').strip()

    if order == '':
        break
```

The preceding code asks the user to enter a topping with `input`, and then invokes `str.strip` on the input string to remove any leading or trailing whitespace. That is then assigned to `order`. If `order` is the empty string, then `break` exits from the `while` loop.

Moreover: every time `while` executes, it checks that there are fewer than three elements in the `added_toppings` list. If the list contains three or more items, then the loop exits.

If the user entered a nonempty string, then the program checks if the user's input is in `toppings`. Remember that `in` on a dict checks for an exact match in the keys, including punctuation, whitespace, and capitalization.

If the value of `order` is a key in the dict, then `price` is assigned the new price, which is then added to `total`. (That could be done in a single step, but it's often easier to understand when broken down.) The user's order is also added to the end of the `added_toppings` list. Then the user is told the current topping's price, the total so far, and what toppings they have already ordered.

You have already seen, in Chapter 6, how `str.join` allows you to get one string from a list of strings. That's perfect for what we want to do here. By using the string `',  '` (i.e., a comma followed by a space) as the "glue" for `str.join`, you'll get a comma-separated list of toppings. That expression can be put directly inside an f-string's `{}`.

```
    if order in toppings:
        price = toppings[order]
        total += price
        added_toppings.append(order)
```

```
        print(f'{order} costs {price}; total is now {total}')
        print(f'Current toppings: {', '.join(added_toppings)}')
    else:
        print(f'Sorry, but {order} is not a valid topping. Try
again.')
```

After the `while` loop has ended — either because there are three toppings, or because the user pressed Enter with a smaller number of toppings — the program prints a final summary:

```
print(f'Total = {total}')
print(f'Toppings = {', '.join(added_toppings)}')
```

And that's it! Here is the full solution to the exercise:

```
total = 10
toppings = {'onion':1, 'garlic':1, 'extra cheese':3,
            'eggplant':2, 'mushrooms':2}
added_toppings = []

while len(added_toppings) < 3:
    order = input('Enter topping: ').strip()

    if order == '':
        break

    if order in toppings:
        price = toppings[order]
        total += price
        added_toppings.append(order)

        print(f'{order} costs {price}; total is now {total}')
        print(f'Current toppings: {', '.join(added_toppings)}')
    else:
        print(f'Sorry, but {order} is not a valid topping. Try
again.')

print(f'Total = {total}')
print(f'Toppings = {', '.join(added_toppings)}')
```

Figure 7-2 shows the visual output from the Python Tutor after running the preceding program.

Print output (drag lower right corner to resize)

```
Enter topping: onion
onion costs 1; total is now 11
Current toppings: onion
Enter topping: garlic
garlic costs 1; total is now 12
Current toppings: onion, garlic
Enter topping: elephant
Sorry, but elephant is not a valid topping. Try again.
Enter topping:
Total = 12
Toppings = onion, garlic
```

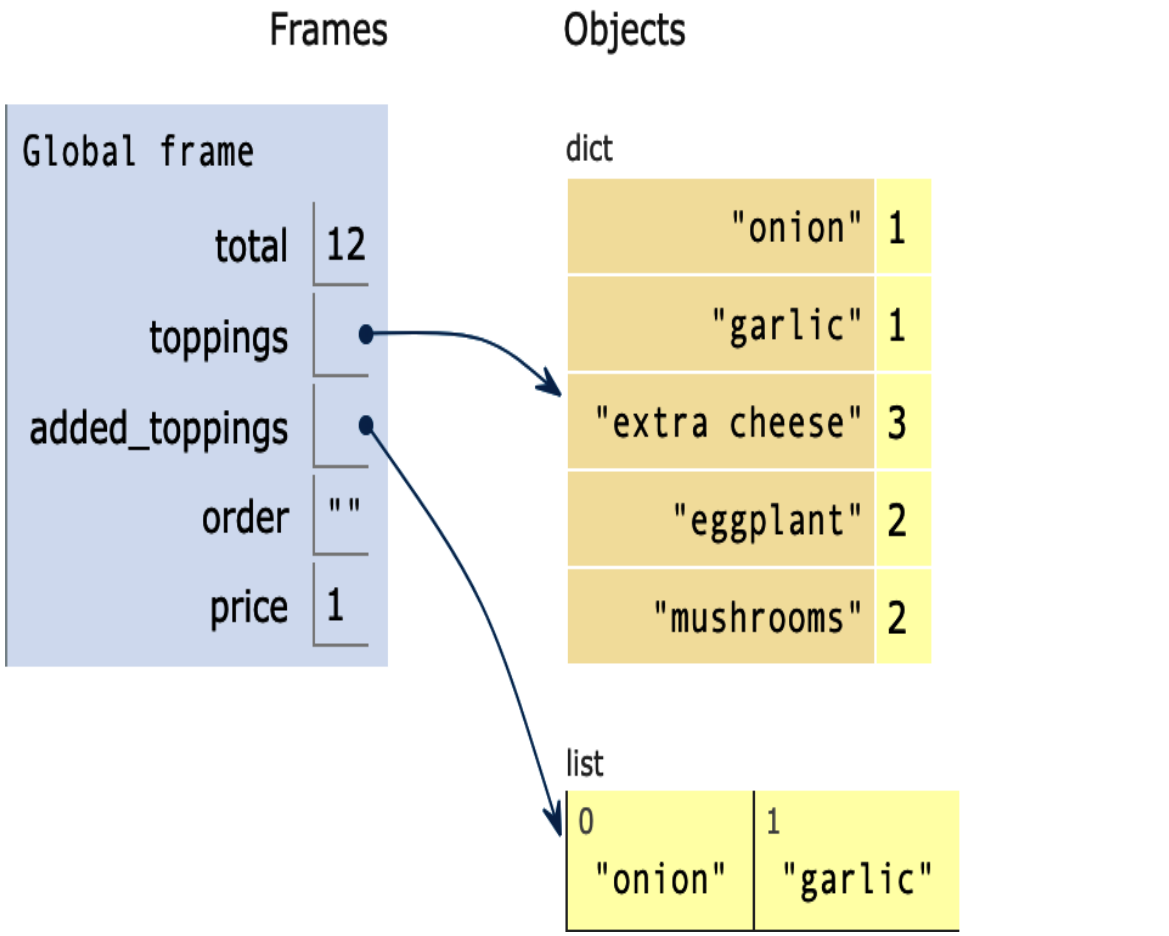


Figure 7-2. Visualization of the pizza program after running

Dicts Are Mutable

In previous chapters, you learned that certain values, such as integers and strings, are *immutable*, meaning that the value cannot change. You can create a new value based on an immutable value, but you cannot change it. You also learned that some other values, such as lists, are *mutable*, meaning that the value can change.

Practically speaking, this means that if you want to add a letter to an existing string, then you need to create an entirely new string, because strings are immutable. But if you want to add an element to a list, then you can just invoke `list.append`. The list's contents will change, but it'll still be the same list.

Like lists, Python's dicts are mutable. This means that we can do three things:

- Replace the value that is already associated with a key
- Add a new key-value pair to the dict
- Remove an existing key-value pair from the dict

In all of these cases, as you'll see, the actions take place via the key. That is, while the values are an important part of any dictionary, you always get to the value via the key.

To replace an existing value in a dict, just assign the new value to the key:

```
d = {'a': 10, 'b': 20, 'c': 30}

d['a'] = 999
print(d)
```

This is similar to replacing a value at a given index in a list. The difference, of course, is that a dict key can be any immutable value. It might seem

strange to have a string inside of the `[]`, but that's normal and necessary if the key is a string.

The preceding code changes the value associated with `'a'` from 10 to 999, as you can see from its output:

```
{'a': 999, 'b': 20, 'c': 30}
```

Remember that there is a big difference between assigning to `d['a']` and `d[a]`. The former assigns to the pair with a key of the string `'a'`. The latter assigns to the pair whose key is in the *variable* `a`.

What if you want to add a new key-value pair? With lists, you needed to invoke the `list.append` or `list.extend` method, so it's normal to think that you would need to use a dict method. But in fact, adding a new pair to a dict is identical to updating the value of an existing key-value pair; just assign to the dict:

```
d = {'a': 10, 'b': 20, 'c': 30}

d['x'] = 500
print(d)
```

Because each key is unique within a dict, assigning a key-value pair can have only one of two effects:

- If the key already exists, then its associated value is updated.
- If the key doesn't exist, then a new key-value pair is added to the dict.

For this reason, the output from the preceding code is:

```
{'a': 10, 'b': 20, 'c': 30, 'x': 500}
```

There is no limit to the number of key-value pairs a dict can contain, and you don't have to tell Python in advance how big you might eventually

expect it to be. So long as your computer has free memory available, you can continue to add new key-value pairs to a dict.

To remove a key-value pair from a dict, use the `dict.pop` method, passing the key as an argument. The pair will be removed, and the value associated with the key you provided is returned, too:

```
d = {'a': 10, 'b': 20, 'c': 30}

value = d.pop('b')
print(f'Removed {value}')    # prints "Removed 20"
print(d)                     # prints {'a': 10, 'c': 30}
```

Passing a nonexisting key to `dict.pop` results in a `KeyError`.

The fact that dicts are mutable means that they can be used to keep track of information over the course of a program. This is quite similar to tracking values with a number of variables. Keeping them in the same dictionary makes it easier to read and write the program, as well as pass, print, analyze, and manipulate the dictionary down the road.

That describes a second common paradigm for dictionary use: you create the dict at the start of the program, setting key-value pairs in which the values are starting defaults. As the program progresses, you update the values, but neither add nor remove keys. For example, here is a program that counts the number of odd vs. even digits in a string:

```
counts = {'odds':0, 'evens':0}

text = input('Enter digits: ').strip()

for one_character in text:
    if not one_character.isdigit():
        print(f'{one_character} is not a digit; ignoring')
        continue

    if one_character in '02468':
        print(f'{one_character} is even')
        counts['evens'] += 1
    else:
        print(f'{one_character} is odd')
```

```
            counts['odds'] += 1

print(counts)
```

Here is the result from the input `12345245`:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
2 is even
4 is even
5 is odd
{'odds': 4, 'evens': 4}
```

In the preceding code, the dictionary both starts and ends with the same keys, `'odds'` and `'evens'`. But the program increments the dict's values as it finds even and odd digits. Notice that because the program doesn't do anything with the digits, the test for an even digit is just `in '02468'`. When the program ends, it prints the dict.

Exercise: Vowels, Digits, and Others (Dict Edition)

In Chapters 5 and 6, you implemented a program that tracked the number of vowels, digits, and others in a user's input string. This time, you'll do the same thing, counting the number of times each type of character appears — but you'll use a dictionary with three key-value pairs to do so.

Specifications

Here's what you need to do:

- Define a dict with three keys, `'vowels'`, `'digits'`, and `'others'`. The value for each key should be 0.
- Ask the user to enter some text, and assign it to the variable `text`.

- Go through the string, one character at a time.
    - Each time you encounter a vowel (a, e, i, o, or u), add 1 to the value for `'vowels'`.
    - Each time you encounter a digit (0-9), add 1 to the value for `'digits'`.
    - In all other cases, add 1 to the value for `'others'`. This includes punctuation and whitespace characters.
- Print the entire dictionary on the screen.

Example:

```
Enter text: hello!! 123

{'vowels': 2, 'digits': 3, 'others': 6}
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The solution to this exercise is similar to that from the previous chapter. The program starts by defining a dict with three key-value pairs, whose values are all 0:

```
counts = {'vowels': 0, 'digits': 0, 'others': 0}
```

The program then continues with the same `if`/`elif`/`else` conditions that you've used before. The difference, though, is what happens inside each of the blocks: rather than increment a variable directly, the program uses `+=` on the value for the appropriate dict key:

```
text = input('Enter text: ').strip()

for one_character in text.lower():
    if one_character in 'aeiou':
        counts['vowels'] += 1
```

```
    elif one_character.isdigit():
        counts['digits'] += 1
    else:
        counts['others'] += 1
```

Finally, you can print a dict — or any Python data structure — just by handing it to `print`:

```
print(counts)
```

Of course, if you are interested in printing the individual values, you can retrieve them individually from the dict:

```
print(f'Vowels: {counts["vowels"]}')
print(f'Digits: {counts["digits"]}')
print(f'Others: {counts["others"]}')
```

NOTE

In the preceding code, the keys are strings, and must thus still be quoted. The most recent versions of Python are fine with single quotes inside `{}`, even when the f-string itself is defined with single quotes, but it's often clearer for the people reading your code to use different types of quotes.

Here's the full solution:

```
counts = {'vowels': 0, 'digits': 0, 'others': 0}

text = input('Enter text: ').strip()

for one_character in text.lower():
    if one_character in 'aeiou':
        counts['vowels'] += 1
    elif one_character.isdigit():
        counts['digits'] += 1
    else:
        counts['others'] += 1

print(counts)
```

It's important to remember that as this program runs, the dictionary doesn't change size. But it still changes, with the values updating with each iteration of the `for` loop, reflecting the number of vowels, digits, and others that were found.

GenAI Task: Interview

Once you have written this program, have a GenAI chatbot interview you about how it works. Here is a sample interaction that I had with Gemini:

```
I'm reading an intro Python book, and am currently in the chapter
about dictionaries. I was asked to write a program that gets the
user's input, and uses a dict to count how many vowels, digits,
and
other types of characters are in that input. Here's the program
that I
wrote:


counts = {'vowels': 0, 'digits': 0, 'others': 0}
text = input('Enter text: ').strip()

for one_character in text.lower():
    if one_character in 'aeiou':
        counts['vowels'] += 1
    elif one_character.isdigit():
        counts['digits'] += 1
    else:
        counts['others'] += 1

print(counts)

The program works, which is great. I want you to interview me
about
this program, asking me questions that force me to demonstrate I
really know what I wrote. Keep asking me questions until I tell
you to
stop--and for each answer, tell me what was right, and what was
wrong.
```

Gemini wrote:

```
That is a very clean and readable script! Using a dictionary to
organize your data is a great habit to get into early on. It
makes the
code much more scalable than just using a bunch of loose
variables.

I'm ready to dive into the interview. I'll ask one question at a
time.
Once you answer, I'll give you feedback and move on to the next
one.

### Question 1

In your code, you have the line: `for one_character in
text.lower():`

**What is the specific benefit of calling `.lower()` right there
in
  the loop header, and how would your results change if you
removed it?**
```

I then responded:

```
Using str.lower forces the user's input to lowercase, which means
that
our count of vowels will include capitalized vowels (i.e., A, E,
I, O,
and U). Without this call to `str.lower`, we either would have
lost
the count of those vowels, or we would have had to check this in
another way, such as using a vowel string of 'aeiouAEIOU'.
```

Gemini then wrote:

```
That is a perfect explanation. You hit the two most important
points:
**normalization** (ensuring 'A' and 'a' are treated the same) and
**logic simplification** (keeping your comparison string short).

### Feedback

* **What was right:** Your understanding of case sensitivity in
Python
    strings is spot on. You correctly identified that `'a' ==
'A'` is
```

```
    `False`, and therefore `lower()` acts as a filter to make the
    membership test (`in 'aeiou'`) work for all cases.

* **What was wrong:** Nothing! Your logic is technically sound.
As a
    tiny "pro-tip," normalization is generally preferred over
checking
    `'aeiouAEIOU'` because it makes the code more resilient if
you
    were to add more complex logic later.

### Question 2

Let's look at how you are updating the data. You wrote:
`counts['vowels'] += 1`

**This line is actually performing two distinct operations on the
  dictionary. Can you explain what those two steps are? Also, if
you
  had forgotten to initialize the dictionary and it was just
empty
  (`counts = {}`), what would happen when the code reached that
line?**

I'm ready for your explanation of the dictionary update process
whenever you are!
```

Iterating Over Dicts

In previous chapters, you saw that Python's built-in data structures are designed to be iterable, meaning that they know how to behave in `for` loops: iterating over a string produces each of the string's characters, one at a time. Iterating over a list or tuple produces each of its elements, one at a time.

What happens if you try to iterate over a dict? Does that even work?

```
d = {'a': 10, 'b': 20, 'c': 30}

for one_item in d:
    print(one_item)
```

This will work — and it will print each of the dict's *keys*:

```
a
b
c
```

As you can see, iterating over a dictionary produces its keys. This shouldn't come as as complete surprise, since you've already seen that every action on a dict is done via its keys. You can take advantage of this to print all of the key-value pairs in a dict:

```
d = {'a': 10, 'b': 20, 'c': 30}

for key in d:
    print(f'{key}: {d[key]}')
```

Sure enough, this prints:

```
a: 10
b: 20
c: 30
```

Some Python programmers think that it's best to iterate over a dict by invoking the `dict.keys` method, which returns a special data structure containing the keys.

This will work, but it clutters your program with an unnecessary method call that slows your code a bit. In other words, using `dict.keys` is almost certainly not worthwhile. Don't be surprised, however, if you see this kind of code in the wild:

```
d = {'a': 10, 'b': 20, 'c': 30}

for key in d.keys():
    print(f'{key}: {d[key]}')
```

There is one small problem with this kind of code, however: it requires that the loop body explicitly reference the dict. That's not a technical problem, per se, but it is a bit ugly. For that reason, I like to iterate over dicts using `dict.items`. That method returns a two-element `(key, value)` tuple

with each iteration. As you've seen before, Python supports unpacking directly in the first line of the `for` loop:

```
d = {'a': 10, 'b': 20, 'c': 30}

for key, value in d.items():
    print(f'{key}: {value}')
```

Once again, the program prints:

```
a: 10
b: 20
c: 30
```

Open-Ended Counting

So far, you have seen two paradigms for using dictionaries:

- Defining a dict at the top of a program and treating it as a read-only database that won't change over the course of the program. You saw this with both the pizza and restaurant programs.
- Defining a dict at the top of a program with keys and initial values. The program neither adds nor removes keys, but does update the values over the course of the program. You saw this with the "odds and evens" and "vowels, digits, and others" programs.

There is, however, a third common paradigm for working with dicts, one that is used for tracking values. That sounds like the second paradigm, except that there is a catch: you don't know in advance just what you'll be tracking. You thus start with an empty dictionary, with neither keys nor values. As you encounter something you want to track, you first check to see if it's in the dictionary. If so, then you update the value to reflect the new item. But if the item isn't yet in the dict, then you add a new key-value pair.

In this third paradigm, the dictionary grows over time, adding new keys as needed. But the values are also updated when you encounter them

additional times.

For example, assume you want to count the number of times each character appears in a user's input. You don't know in advance what characters the user will enter. You could define a dict, `counts`, in which the keys are every character known to Python, and the values are all 0. Then you can iterate over the string with a `for` loop, executing `counts[one_character] += 1` for each character.

However, this solution is completely impractical: creating such a dictionary would take time, and would consume a huge amount of memory. And for what? You won't be using the overwhelming majority of those characters, which will be in languages you don't use, or even emojis.

Rather, you can define an empty dictionary, `counts`, and then increment the count for the character you're currently examining. However, if you merely say `counts[one_character] += 1` for each character, you will quickly get an error. That's because `counts[one_character] += 1` assumes that `one_character` is already a key in `counts`, and that you can add 1 to its current value. The first time you encounter `one_character`, though, it isn't yet a key in the dict, and doesn't have a value.

You thus need to check whether the character is already a key in the dict:

- If `one_character` is already in `counts`, increment it with `+=`, saying `counts[one_character] += 1`.
- If this is the first time you're encountering `one_character`, then add a new key-value pair to the dict saying `counts[one_character] = 1`.

Here's an example of such a program:

```
counts = {}

text = input('Enter text: ').strip()

for one_character in text:
```

```
    if one_character in counts:       # have we seen this
character before?
        counts[one_character] += 1    # ... great, increment its
count by 1
    else:
        counts[one_character] = 1     # ... we haven't? add a new
key-value pair

for key, value in counts.items():     # Go through each key-value
pair
    print(f'{key}: {value}')          # Print the key and value
```

Here is an example run of the preceding program:

```
Enter text: this is an amazing example program
t: 1
h: 1
i: 3
s: 2
 : 5
a: 5
n: 2
m: 3
z: 1
g: 2
e: 2
x: 1
p: 2
l: 1
r: 2
o: 1
```

Notice that every character in the user's input is counted, including spaces, punctuation, and emojis, as you can see from the following run:

```
Enter text: I am happy. 😊
I am not sad. 😯
I: 2
 : 8
a: 4
m: 2
h: 1
p: 2
y: 1
.: 2
```

```
😊: 1
n: 1
o: 1
t: 1
s: 1
d: 1
😳: 1
```

Exercise: Counting Words

The preceding example showed how you can use dictionaries to track and count the number of times each character appears in a text string. In this exercise, you will ask the user to enter multiple text strings, stopping only when they press ENTER (i.e., enter an empty string). Use a dictionary to track how often each word appears in the user's entries. When the user stops entering new text, print each word and the number of times it appeared. We can assume that the user will only enter lowercase letters and spaces, without any capital letters, punctuation, or emojis.

Specifications

Here are the specifications for this exercise:

- Define an empty dict, `counts`.
- Ask the user repeatedly to enter a text string. We can assume that the string will contain all lowercase letters and no punctuation or emojis.
- If the user enters an empty string, or one containing only whitespace, then stop asking.
- Go through each word in the user's input.
    - If the word is already a key in `counts`, then increment its count by 1.

    - If the word doesn't yet appear in `counts`, then add the key (the word) and value (1).
- After finishing with the user's input, print each word and its count on a separate line.

For example:

```
Enter text: this is the first run of this program
Enter text: the second run will undoubtedly be even better
Enter text: this is very exciting
Enter text:

this: 3
is: 2
the: 2
first: 1
run: 2
of: 1
program: 1
second: 1
will: 1
undoubtedly: 1
be: 1
even: 1
better: 1
very: 1
exciting: 1
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

There are four parts to this program.

The first part is also the easiest to understand, and the most similar to what you've written in previous exercises. You set up the empty `counts` dict, `{}`. Then you need an infinite loop, implemented with `while True`, because you don't know in advance how many inputs you will get from the user. If the user enters an empty string, then the program uses `break` to exit from the `while` loop:

```
counts = {}

while True:
    text = input('Enter text: ').strip()

    if text == '':
        break
```

Inside the `while` loop, the program needs to break `text` into individual words, and consider each of them in turn. This requires using the `str.split` method, which returns a list of strings. And then it requires using a `for` loop to go over each word in turn:

```
    for one_word in text.split():
```

Next is the heart of the program, where it counts the word. Ideally, you might want to just say `counts[one_word] += 1`. However, this will only work once `one_word` is already a key in `counts`. Before then, running this code will result in a `KeyError`. And so the program needs to check whether `one_word` is already a key in `counts`, using the `in` operator:

- If `one_word in counts`, then the program can just run `counts[one_word] += 1`, incrementing the count for that word.
- In all other cases, the program adds `one_word` as a new key, with `1` as its associated value.

The first time the program encounters a word, it'll add it with a count of 1. On subsequent runs, it'll add 1 to the current number, and store that as the new value:

```
        if one_word in counts:      # have we seen this word
before?
            counts[one_word] += 1   # ... great, increment its
count by 1
        else:
```

```
            counts[one_word] = 1     # ... we haven't? add a new
key-value pair
```

Finally, once `counts` has been fully populated with words and counts, the program iterates over every key-value pair, thanks to `dict.items`, printing each key and value:

```
for key, value in counts.items():     # Go through each key-value
pair
    print(f'{key}: {value}')          # Print the key and value
```

Here is the final version of the program:

```
counts = {}

while True:
    text = input('Enter text: ').strip()

    if text == '':
        break

    for one_word in text.split():
        if one_word in counts:       # have we seen this word
before?
            counts[one_word] += 1    # ... great, increment its
count by 1
        else:
            counts[one_word] = 1     # ... we haven't? add a new
key-value pair

for key, value in counts.items():     # Go through each key-value
pair
    print(f'{key}: {value}')          # Print the key and value
```

GenAI Task: Add a Bug

Once you have finished writing this program, I want you to break it by adding a bug — and seeing if the GenAI chatbot can find the bug that you added. The point isn't to see if you can fool AI (though that would be nice!), but to see if the bug you insert breaks things in the way you predict — and if AI will predict the same issue that you created.

Here is an example exchange that I had with ChatGPT:

```
I'm reading an intro Python book, and just finished writing an
exercise that counts the number of words in a user's input. The
user
can enter text as many times as they want; when they enter an
empty
string, the program then prints how often each word appeared. The
assumption is that they will enter only lowercase letters and
spaces,
without other punctuation or characters.

Here's the program that I wrote for this exercise. However, I've
added
a bug to the program. Can you find it? What will go wrong if I
run
this version instead of the regular, working version?

counts = {}

while True:
    text = input('Enter text: ').strip

    if text == '':
        break

    for one_word in text.split():
        if one_word in counts:      # have we seen this word
before?
            counts[one_word] += 1   # ... great, increment its
count by 1
        else:
            counts[one_word] = 1    # ... we haven't? add a new
key-value pair

for key, value in counts.items():    # Go through each key-value
pair
    print(f'{key}: {value}')         # Print the key and value
```

ChatGPT replies:

```
Yep — the bug is on this line:

```python
text = input('Enter text: ').strip
```
```

````
You forgot the parentheses after `strip`. So instead of
**calling**
`strip()` and getting back a cleaned-up string, you're assigning
`text` to the **method object** `strip` (a function attached to
the
string).

### What goes wrong when you run it?

1. `text` is **not a string** — it's a "built-in method" object.

2. This check will never be true:

   ```python
   if text == '':
   ```

   because a method object is never equal to the empty string. So
the
   loop won't stop even when the user presses Enter on an empty
line.

3. Then you hit:

   ```python
   for one_word in text.split():
   ```

   But method objects don't have `.split()`. So the program will
crash
   on the first iteration with an error like:

* `AttributeError: 'builtin_function_or_method' object has no
  attribute 'split'`

### The fixed line

```python
text = input('Enter text: ').strip()
```
````

I then tried again:

```
Good job! You found it!
```

```
Let's try again. What bug did I insert here?

counts = {}

while True:
    text = input('Enter text: ').strip()

    if text == '':
        break

    for one_word in text.split():
        if one_word in counts:      # have we seen this word
before?
            counts['one_word'] += 1   # ... great, increment its
count by 1
        else:
            counts['one_word'] = 1    # ... we haven't? add a new
key-value pair

for key, value in counts.items():    # Go through each key-value
pair
    print(f'{key}: {value}')         # Print the key and value
```

ChatGPT's response:

```
The bug is that you put quotes around `one_word` when using it as
a dictionary key:

```python
counts['one_word'] += 1
# and
counts['one_word'] = 1
```

That means you're **always** updating the key literally named
`"one_word"` (a single string), instead of using the variable
`one_word` (which changes for each word).

What will go wrong?

* Your dictionary will end up with **only one key**:
`"one_word"`.
* Its value will be the **total number of words entered** (across
all
  lines), not counts per word.
```

````
Example input:

```
hello hello world
```

Expected:

* `hello: 2`
* `world: 1`

What your buggy version prints:

* `one_word: 3`

### Fix

Remove the quotes:

```python
counts[one_word] += 1
counts[one_word] = 1
```
````

How Do Dicts Work?

Because keys can be of many different types, and aren't restricted to integers, dicts offer a straightforward way to map from the real world to a data structure. And indeed, in the Python world dictionaries are often called *mappings* for that reason.

But dicts are also important for another reason: finding a key, or retrieving a value via a key, is extremely fast. That's because dictionaries make use of a "hash function," an algorithm that returns a close-to-unique value for each input. When you store `d['a'] = 100`, Python runs `hash('a')`, gets a number back and uses that number to store the key-value pair, `'a'` and `100`. When you then store `d['b'] = 200`, Python runs `hash('b')`, gets a number back, and uses that number to store the key-value pair, `'b'` and `200`.

How does this make dicts so fast? Well, consider that you then run `'a' in d`. Python (again) runs `hash('a')`, gets a number back, and uses it to check whether `'a'` is already being used as a key. If so, then it returns `True`. If not, then it returns `False`. If you search through a 1-million element list for a particular value, you might need to go through every element to find what you're looking for. With a dict, Python can jump directly to where the hash function points it, and find out if the key is there.

Things are a bit more complex than this, but not by much. For example, the dict implementation needs to handle situations in which more than one key-value pair might be assigned to a particular location in memory. This situation, known as a *collision*, is part of the implementation of Python's dicts, meaning that we don't need to worry about it.

This means that when you search for a key in a dict, you are getting the fastest possible search time in Python. And when you retrieve a value from a dict based on a key, you're getting the fastest retrieval time that Python can offer. This is why Python itself, under the hood, uses a dictionary to store all of your variables and values: variables are stored as strings, and values are stored as themselves. Retrieving the value of the variable `x` basically has Python translate `x` into a string, then retrieve the corresponding value from a dict.

This explains why dict keys are unique within a dictionary: if this were not the case, then it would be impossible for Python to know which one it should retrieve.

But you might still be wondering why Python only allows for immutable types to be used as dict keys. This is especially true if you have used other programming languages, where dict-like data structures typically don't have this restriction.

The reason for this rule is because Python stores a key-value pair based on the result of invoking `hash` on the key. If the key were to change, then the hash result would be different — meaning that Python would be unable to find a key, let alone retrieve its value. There are languages in which modifying a key results in "rehashing" the dictionary, recomputing the

locations for each of the key-value pairs. To avoid this sort of problem, Python restricts what can be used as a key.

Technically speaking, the term "hashable" refers to data structures on which the `hash` function can be invoked, and for beginning Python programmers there is a very large overlap between that and "immutable." That said, the two terms aren't identical — but the differences tend to come up rarely, and aren't worth getting into at this stage of the game.

Summary

In this chapter, you learned about dictionaries: not only their syntax, but three paradigms for their usage in your own programs. You saw how whenever you have a unique identifier for a data structure, it can be a good idea to turn it into a dictionary, with the unique identifiers serving as the keys. Finally, you learned how dicts work, and how you can (and should) integrate them into your programs.

Believe it or not, you have now learned the key, fundamental data structures in Python. Everything else in the language is built on the infrastructure of integers, floats, strings, lists, tuples, and dictionaries. There are other data structures, but they are either based on these six or are described in their terms. Becoming familiar, and even fluent, with these data structures is the key to true Python fluency.

In the next chapter, you'll learn about how to read from and write to files — allowing you to store data across program invocations, transfer data to other computers, and share data with colleagues and friends.

Chapter 8. Files

A program consists of two things: code and data. As you've seen, the data is stored in a combination of structures — numbers, strings, lists, tuples, and dictionaries. The code's job is to read from and write to that data.

What happens to the data when the computer is shut off or rebooted? Or if you want to take the data and share it with someone else? Or if you want to back up the data, in case something goes wrong with the program?

These are just three examples of why the idea of a *file* is so important: it allows you to export and store data structures from the computer's memory. Later on, you can have the program recreate the data structures, exactly as they were at the time the file was created. You probably never thought of Word, PDF, or PowerPoint files as snapshots of data structures, but that's precisely what they are.

While you're editing a spreadsheet, your data exists inside the Excel program, as a combination of data structures. When you save that spreadsheet to a file, you're creating a record of the data structures, including their contents, that are in the computer's memory. You can load that spreadsheet into Excel at a later time, or transfer it to another computer and load it there.

Python makes it easy to read from and write to files. That's particularly true in the case of plain-text files, which are commonly used in configuration and logging. In this chapter, you'll learn how to read from and write to files, how to translate from a file's contents into data structures, and also how to do the reverse, saving data structures to plain-text files on disk. You'll also learn the preferred ways to accomplish these tasks, and why they are considered more idiomatic.

Reading from a File

What does a program need to do, if it wants to read from a file? Back in the early days of the computer industry, it was normal for a computer to make direct contact with a disk (often known more generally as a *filesystem*), and read the data it wanted from the file. There are numerous problems with this approach, starting with the unreasonable need for every program to know how to work with every kind of filesystem, and including the lack of security and permissions that were common in those days.

Modern computers avoid these issues by having all communication with a filesystem go through the operating system. If a program wants to read from a file, it tells the operating system — e.g., Windows, macOS, or Linux — what file it wants to access. If the operating system agrees that the user should be allowed to read from that file, it provides the program with a *file handle*, a sort of software mediator, through which the file can be accessed. The operating system, and not the individual program, handles issues of compatibility and security, eliminating the need for each programmer, and each program, to worry about such things.

The process of getting a file handle is traditionally known as *opening* a file. When the file handle is no longer needed, the program then *closes* the file. Once the file is closed, the file handle is no longer useful; using the file handle to access the file will result in an error.

A program that uses a file will typically have this general structure:

- Open the file
- Read some or all of the file into memory
- Turn the data into data structures
- Produce output, either on the screen or into a separate file
- Close the file

What sorts of data structures do you create from the file's contents? The sky's the limit, and every program does things a bit differently. However, reading data from a text file, which is what you'll learn to do in this chapter, means working with strings. Often, you'll want to take each string and break

it apart using `str.split`, as you saw in Chapter 6. You can then do many things with that broken-apart string, from creating new lists to creating dictionaries or even complex combinations of lists, tuples, and dictionaries.

Opening a File

The first task in working with a file, as the preceding section indicated, is to "open" the file. In Python, you can do this with the `open` function, which looks like this:

```
f = open('filename.txt')
```

`open` takes a string as an argument. That string contains the name of the file you want to open. Later in this section, you'll learn about the variety of filenames that you can provide, but for now you can assume that the file you're opening is in the same directory, or folder, as the program you're running.

The return value from `open`, assigned to `f` in the preceding code, is a *file object*, Python's version of the "file handle" you read about. Actually, the official Python documentation doesn't talk about "file objects" so much as *file-like objects*, since `open` can return a variety of different values, depending on how it is invoked. These values all behave in similar ways, and all give a program access to a file, thus giving them the moniker "file-like object." And while "file-like object" might be more precise, this book uses the term "file object," which gets the point across while feeling a bit friendlier and less bureaucratic.

By default, `open` allows you to read from a file, but not write to it. Many coders prefer to make it explicit that they wish to read from a file by including an optional second argument — the string `'r'`, short for *read* — to `open`:

```
f = open('filename.txt', 'r')
```

Later in this chapter, you'll learn how to write to a file by passing a different second argument to `open`. The fact that read-only mode is the default for `open` indicates that it is more common to read from a file than to write to one. Opening a file for reading is also safer, removing the chance that you'll accidentally remove a file or change its contents.

> **NOTE**
>
> Trying to `open` a file for reading when the file does not exist will result in an error, known formally as `FileNotFoundError`. If you get this error, make sure that the spelling and capitalization of the filename you provided are accurate. You might also have made a mistake with the file's location; see "Specifying Filenames" for details and hints. Other common errors are `IsADirectoryError`, if you try to open a directory as if it were a file, and `PermissionError`, if you try to open a file to which you don't have access.

You can ask Python to print the file object with `print(f)`, which will produce something like the following:

```
<_io.TextIOWrapper name='filename.txt' mode='r' encoding='UTF-8'>
```

From this output, you learn the following:

- The file object's formal type is `_io.TextIOWrapper`. Simply put, this means that you are reading from a plain-text file. This is one example of a "file-like object," where the type's real name can be a confusing string of jargon and abbreviations.
- The file object knows the name of the file from which it's reading. That is specified as a string — in this case, *filename.txt*.
- This file object is in read-only mode. It can read from the file, but cannot modify it in any way.
- The file's text can contain any character from the international Unicode standard, which basically means that it can contain any character from any language, as well as emojis, symbols, and

musical notation. Those characters should be written using a commonly supported format known as UTF-8, which is common on many computers.

To close the file, you simply invoke the `close` method on the file object:

```
f.close()
```

What happens if you don't close a file? In many cases, nothing bad: when a Python program exits, it will close the file as part of its shutdown routine. However, if the program runs for a while, that might not happen for some time. Even so, the operating system has a limited number of file handles it can provide to running programs, and they can run out if the computer is particularly busy. For this reason, it's considered polite and a best practice to close the file. Later in this chapter, you'll learn about the `with` keyword, and how it helps to close files automatically.

Specifying Filenames

In order to open a file, you'll need to specify a filename, typically a string referring to a file. If the program and file are both in the same directory or folder, then nothing special is needed beyond naming the file. If the file is called *myfile.txt*, then you can simply tell Python `open('myfile.txt')` and it'll work.

NOTE

When you create a document on your computer, you likely give it a name that describes its contents, such as *budget* or *mafiahit*. But the filename includes a second part, typically after a `.` character, known as the *extension*. Common extensions are *docx* for Microsoft Word, *pdf* for PDF, *xlsx* for Excel, *txt* for text, and *py* for Python. The full names of the documents might thus be *budget.xlsx* or *mafiahit.pdf*, indicating that they are Excel and PDF documents, respectively. However, these extensions are hidden by many operating systems. You might want to enable the "show file extensions" option on your computer, in order to see the complete names for these documents. Remember that opening a file requires its full name; if you do not include its extension, then Python will fail to open the file, giving you a "no such file" error message.

But what if the program and file aren't in the same directory? What if you aren't sure if they are in the same location? That's where things can get complex, especially for modern computer users who are used to clicking on folders and filenames, rather than writing them out.

To keep things simple, and not end up writing an entire treatise on writing relative pathnames, try to use *absolute pathnames* as much as possible. On Windows, these will likely start with `c:\` and continue with one or more folder names, separated by `\` characters, ending with a filename. Some examples:

- *c:\users\reuven\Desktop\myfile.txt*
- *c:\users\reuven\Documents\pythoncourse\myfile.txt*

As you learned in Chapter 4, Python interprets a backslash (`\`) as the start of a special character, as in `'\n'` or `'\t'`. To avoid confusing Python, values involving Windows paths should be defined with "raw strings," meaning an `r` before the opening quotes. Defined as variables in a Python program, the preceding paths would be written as:

- `filename = r'c:\users\reuven\Desktop\myfile.txt'`
- `filename = r'c:\users\reuven\Documents\pythoncourse\myfile.txt'`

In your Python program, you can then say `open(filename)`.

If you're running macOS or Linux, then paths are different in two ways: first, Unix uses `/` to separate directory names from one another. This means that you don't need to go through the raw-string or backslash-doubling gymnastics mentioned in Chapter 4. Also, Unix paths start with `/`, indicating the "root" of the filesystem, roughly equivalent to `c:\`. Here are two Unix absolute paths defined as strings:

- `filename = '/Users/reuven/Desktop/myfile.txt'`

- `filename = '/Users/reuven/Documents/pythoncourse/myfile.txt'`

You can download a ZIP file containing files for this book from *https://files.lerner.co.il/exercise-files.zip*. Unzip that file into a known location, such as your `Desktop` folder, and you will be able to access these files in this chapter's exercises.

> **NOTE**
>
> This ZIP file contains five short text files that I have used for many years: *mini-access-log.txt*, *wcfile.txt*, *shoe-data.txt*, *nums.txt*, *passwd.txt*, *macos-passwd.txt*, and *linux-etc-passwd.txt*.

The read Method (and Why It's Usually a Bad Idea)

Once you have opened a file, how can you read from it? If you look through Python's documentation, then you might discover the `read` method. On the face of it, this seems like it would fit the bill: after opening the file, then you can invoke `read` to get its contents back as a string. And indeed, that will work:

```
f = open('myfile.txt')
print(f.read())
f.close()
```

This will, generally speaking, do what you want, printing the contents of *myfile.txt* on the screen. But there are some issues with it. For starters, this doesn't store the file's contents in any variable, but directly prints it onto the screen. It's probably a good idea to read it into a variable, rather than just passing the contents along to `print`:

```
f = open('myfile.txt')
text = f.read()
print(text)
f.close()
```

This is better, but the odds are good that you don't want to just print the entire file. Rather, the file is probably structured in a certain way, such that you can examine its contents and make decisions about what you want to print and what you want to ignore. You can, of course, examine and break the string apart, perhaps using `str.split`, but that's just more work for you, and it would be nice to avoid it.

The biggest problem is also the rarest: what if the file contains 10 *terabytes* of data, where 1 terabyte is the same as 1,000 GB? Python will try to read that entire file into memory as one humungous string, and will run out of memory in doing so. Your program will fail, and it's possible that your computer will slow down significantly or even crash when trying to allocate such a large amount of memory.

For this reason, it's rare to read a file all at once with the `read` method.

One alternative is to invoke `read` with an integer argument, the maximum number of characters you are willing to receive. Ideally, `read(1000)` will return a string containing the next 1,000 charactres from the file, starting wherever the invocation of `read` ended. (The first time you invoke `read`, it starts from the beginning of the file.) If you are too close to the end of the file to get all 1,000 characters, `read(1000)` will return a shorter string, with as many characters as it can supply. If you are already at the end of the file, then `read` returns the empty string.

This technique removes the risk of reading too much into Python all at once. However, it introduces a different problem, namely that text files are generally structured on a per-line basis. When you invoke `read(1000)`, then you'll get the next 1,000 characters, likely including some `'\n'` (newline) characters that indicate line breaks. This can be frustrating to overcome.

Python does provide a `readlines` method, which returns the contents of the file as a list of strings, with each line of the file in a separate string. But if the file is too big to be read into a single string, then it's definitely too big to read into a list of strings.

The solution to this dilemma is surprisingly elegant: if you iterate over a file object in a `for` loop, each iteration brings the next line in the file. And by "the next line," I mean that Python retrieves all of the characters through the next `'\n'` (newline) character. This provides the best of all worlds: you won't overload memory, because each iteration only reads a single line into a string, and it's highly unlikely that a single line would overwhelm your computer's memory. Granted, you don't get the entire file into memory at the same time, but you rarely need that.

Here's what it looks like to iterate over (and print) the contents of a file in this way:

```
f = open('myfile.txt')

for one_line in f:
    print(one_line)

f.close()
```

If there is a file named *myfile.txt* in the same directory as Python, then the preceding program will print it, line by line, onto the screen. And when the `for` loop reaches the end of the file, it finishes and program execution continues immediately after the loop body.

NOTE

Calling the loop variable `one_line` reflects the fact that files will *always* return one line, as a string, in each iteration. The loop would work precisely the same way regardless of the variable's name. Choosing good variable names can be hard — but the right choices can make your program easier to read, understand, and maintain.

There is actually one small problem with the preceding program: `one_line` contains the next line from the file, defined as "up to and including the next `'\n'`." This means that `one_line`, by definition, will always end with a `'\n'`. That's not bad, but combined with the fact that

`print` always adds a newline to whatever it prints, it means that file will be printed double spaced, with a blank line between each printed line.

One way to avoid this is to invoke `str.strip` on `one_line` before printing it. `str.strip` returns a new string, one without any whitespace characters — space, newline, or tab — at the start or end:

```
f = open('myfile.txt')

for one_line in f:
    print(one_line.strip())

f.close()
```

Note that `str.strip` will also remove whitespace at the start of the string. If you want to avoid that, and only remove whitespace on the right side, you can use `str.rstrip`, which does just that:

```
f = open('myfile.txt')

for one_line in f:
    print(one_line.rstrip())

f.close()
```

The preceding version of the program will go through the named text file (*myfile.txt*, in this case). `print` adds a newline to its output — but because the output doesn't end with a newline (thanks to `str.rstrip`), the file is printed in its original form, without any new blank lines introduced.

NOTE

What if you try to use the preceding code with a nontext file, such as PDF or Excel? Python will exit with an error, telling you that it cannot read this binary file into a string. There are ways to read binary files in Python, but this book ignores such files, both because they're more complex to handle and because you're less likely to deal with them from within your Python code.

Exercise: File Size

In this exercise, you'll calculate the size — that is, the number of characters — in a file and print it on the screen. You will probably want to use one of the files that comes in the ZIP file you downloaded earlier.

Specifications

Here is what you need to implement for this exercise:

1. Ask the user to enter the name of a file, and assign it to `filename`.
2. Set `total` to 0.
3. Iterate through the file, adding the length of the current line — counting all characters, including whitespace and punctuation — to `total`.
4. When you're done iterating over the lines of the file, print `total`.

Here is an example run of the program, using the *passwd.txt* file from the ZIP file:

```
Enter a filename: passwd.txt
Total size: 9064
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

Solving this problem requires that you remember that when you iterate over a file, you're getting each line as a string. As you've already learned, Python's `len` function returns the length of a string. For that reason, you can start with the skeleton program you've already seen for iterating over a file and printing its contents, changing it to get `filename` from the user and setting `total` to 0:

```
total = 0
filename = input('Enter a filename: ').strip()
f = open(filename)

for one_line in f:
    print(one_line.rstrip())

f.close()
```

Instead of printing each line, though, you want to measure its length. Here's a version of the program that iterates over each line and prints its length on the screen:

```
total = 0
filename = input('Enter a filename: ').strip()
f = open(filename)

for one_line in f:
    print(len(one_line))

f.close()
```

The exercise specifications didn't ask you to print the length of each line. Rather, you were supposed to add the length of each line to `total` and then print the final value of `total`. This program will do that:

```
total = 0
filename = input('Enter a filename: ').strip()
f = open(filename)

for one_line in f:
    total += len(one_line)

f.close()
print(f'Total size: {total}')
```

The program prints:

```
Total size: 9064
```

AI Challenge: Interview

Working with files can be tricky: between getting the filename right, opening the file, iterating over it, handling the string that it returns with each iteration, and closing it, there are a number of new ideas to absorb, as well as syntax that's easy to get wrong. For those reasons, the GenAI task this time is to take your working program and have your favorite GenAI system interview you about it. Here's a transcript of such an interview I did with Claude:

```
I've just written the following as part of an introductory book
on
Python programming:

total = 0
filename = input('Enter a filename: ').strip()
f = open(filename)

for one_line in f:
    total += len(one_line)

f.close()
print(f'Total size: {total}')

Interview me about this program, asking me questions about it.
Each
time I answer, tell me what I got right and wrong. Keep asking
questions until I tell you to stop.
```

Claude says:

```
Great! Let's dive into understanding this program. I'll start
with a
fundamental question:

**Question 1:** What does this program do overall? Walk me
through
  what happens when someone runs it.
```

I respond:

```
This program asks the user to name a file. It then goes through
the
file, one line at a time, adding the size of the current line to
```

```
the
"total" variable. When the loop exits, it prints the total size
of the
file.
```

Claude says:

```
**What you got right:** Excellent! You've correctly identified
the
  overall flow - the program asks for a filename, iterates
through the
  file line by line, accumulates something in the `total`
variable,
  and then prints the result.

**What needs clarification:** You said it calculates "the total
size
  of the file," but let's be more precise. What exactly is
  `len(one_line)` measuring? Is it truly the file size, or is it
  something else? Think about what a "line" actually is when
Python
  reads it from a file.

**Question 2:** When you iterate through a file with `for
one_line in
  f:`, what exactly does each `one_line` contain? Does it include
  anything besides the visible text on that line?
```

Turning Text into Data

There's nothing wrong with reading text from a file and treating it as text. But it's more interesting to take the text in a file and turn it into a data structure that you can then analyze. Sometimes, you'll want to use all of the text in creating that data structure. Sometimes, though, you'll want to be selective about it.

For these examples and exercises, you will use Unix-style *passwd* files, typically stored in */etc/passwd* on Unix, Linux, and macOS systems. (The ZIP file comes with three example versions of these files named *passwd.txt*, *macos-passwd.txt*, and *linux-etc-passwd.txt*.) I have long used *passwd* files

because they are common, relatively easy to work with, and can be turned into a variety of data structures.

> **NOTE**
>
> From the name, you might expect that *passwd* files contain users' passwords. For obvious security reasons, passwords were moved from *passwd* files many years ago, into more secure locations. However, the rest of the file has remained intact, and is used to this day for user management on Unix-type systems.

Each line of a *passwd* file contains information about one user on the system, using a format that looks like this:

```
root:*:0:0:System Administrator:/var/root:/bin/sh
```

If this line, or the entire file, looks weird to you, that's just fine! It isn't meant to be read by people, but rather by programs. A program can read through the file, get the information it needs about a user, and then use that information when it executes.

The `exercise-files.zip` file that you can download includes several versions of */etc/passwd*, including one called *passwd.txt*. Each line contains seven fields, separated by `:` characters:

- The username (`root`)
- A `*` where, years ago, the password was stored
- The user ID number
- The user's group ID number
- The user's name, as displayed to other people
- The user's home directory
- The program that is executed when the user logs in, often known as a "shell"

Given this knowledge, how could you display the username and user ID for each person on this system?

The key to solving this problem is to understand that each line of the field is a string, with fields separated by `:` characters. By invoking `str.split` on the line, you get a list of strings — one in which each element is a separate field. You can then retrieve whichever of those fields you want via a numeric index into the returned list.

For example:

```
filename = 'passwd.txt'
f = open(filename)

for one_line in f:
    fields = one_line.split(':')
    username = fields[0]
    user_id = fields[2]

    print(f'{username}: {user_id}')

f.close()
```

The preceding code works just fine with *passwd.txt*.

But another file in the same ZIP file, *macos-passwd.txt*, is a version of *passwd.txt*, but from my macOS-based computer. They differ only in that Apple includes comments (i.e., lines starting with `'#'`) in its version of */etc/passwd*. The preceding code will break on such a file. Fortunately, an `if` statement inside the `for` loop body will take care of that, allowing the program to handle variants containing comments:

```
filename = 'macos-passwd.txt'
f = open(filename)

for one_line in f:
    if one_line[0] == '#':    # comment line?
        continue              # go onto the next line

    fields = one_line.split(':')
    username = fields[0]
    user_id = fields[2]
```

```
        print(f'{username}: {user_id}')

    f.close()
```

Similarly, some versions of */etc/passwd* have blank lines, or lines containing only whitespace. A version of such a file is in *linux-etc-passwd.txt*. Running `str.strip` on such a blank line returns the empty string, lending itself to a second, similar `if` statement inside the `for` loop:

```
filename = 'linux-etc-passwd.txt'
f = open(filename)

for one_line in f:
    if one_line[0] == '#':    # comment line?
        continue              # go onto the next line

    if one_line.strip() == '':    # nothing left after stripping?
        continue                  # go onto the next line

    fields = one_line.split(':')
    username = fields[0]
    user_id = fields[2]

    print(f'{username}: {user_id}')

f.close()
```

When run, the preceding program goes through *linux-etc-passwd.txt*. It ignores blank lines and comment lines, and prints each username and user ID number.

Printing that information on the screen is nice. But you can also take the data from a text file and turn it into a data structure.

For example, you could use the file to create a dictionary in which the keys are usernames and the values are user ID numbers. If your program will be querying usernames on a regular basis, then having such a dict in memory, and always available, is quite useful.

The strategy here is to create an empty dict, then populate it with one new key-value pair from each line from the file:

```
users = {}                    # empty dict

filename = 'linux-etc-passwd.txt'
f = open(filename)

for one_line in f:
    if one_line[0] == '#':    # comment line?
        continue              # go onto the next line

    if one_line.strip() == '':    # nothing left after stripping?
        continue                  # go onto the next line

    fields = one_line.split(':')
    username = fields[0]
    user_id = fields[2]

    users[username] = user_id     # add a key-value pair

f.close()

print(users)
```

Here's another, fancier option: create a dict whose keys are the "shells" the program runs when a user logs in, and the values are lists of usernames who use that shell. For example:

```
shells = {}                    # empty dict

filename = 'linux-etc-passwd.txt'
f = open(filename)

for one_line in f:
    if one_line[0] == '#':    # comment line?
        continue              # go onto the next line

    if one_line.strip() == '':    # nothing left after stripping?
        continue                  # go onto the next line

    fields = one_line.split(':')
    username = fields[0]
    one_shell = fields[-1].strip()     # grab the shell, remove
whitespace

    if one_shell in shells:                 # have we seen this
shell before?
        shells[one_shell].append(username)  # add user to this
```

```
shell's list

    else:                                       # first time seeing
this shell?
        shells[one_shell] = [username]      # add a 1-element
list to start

f.close()

# iterate through the dict, printing it nicely
for shell, user_list in shells.items():   # get key, value pairs
with dict.items
    print(shell)

    for one_user in user_list:    # nested for loop, iterating
over the list
        print(f'\t{one_user}')
```

This program takes advantage of the fact that a dict's values can be anything, including a list. Here, each key is a string, the name of a shell. But the values are lists of strings:

- The first time that the program encounters a shell, it creates a list with the username as its only element, and assigns that to be the value.
- If it's not the first time, then the program assumes the value is a list, and invokes `list.append`, adding the new username to the existing list.

The end of the program uses a `for` loop to iterate over the dictionary, printing each key-value pair with `dict.items`. Since each value is a list of strings, the program then uses a *nested* `for` loop (i.e., one `for` loop inside another) to print each username on a line by itself. The `'\t'` (tab) character at the start of the string indents each username.

Turning a text file into an in-memory data structure involves using `str.split` to turn the line into multiple fields, `for` loops on the line or the result of splitting it, and then creating a list or dict with some or all of those fields. The fact that you can treat a string as a whole, as a collection

of characters, or as a collection of words gives you a lot of leeway in how to use, break up, and reuse the string.

Exercise: File Analysis

In this exercise, you'll analyze a text file, producing a report for the user on a variety of aspects of the file.

Specifications

Ask the user to enter the name of a text file. Iterate over the file, one line at a time. Along the way, keep track of:

- Number of blank lines (i.e., containing only a `'\n'` character)
- Number of capitalized words
- Number of numbers (i.e., words that only contain digits 0-9)
- Number of punctuation marks (i.e., `'.'`, `'?'`, and `'!'`)
- Number of words that start and end with the same letter (ignoring case)

Keep track of these in the `analysis` dict, which you'll set up before opening and iterating over the file. After reading through the file, print each key-value in `analysis`.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The program starts by getting the filename from the user, opening the file, and setting up the `analysis` dict:

```
filename = input('Enter filename: ').strip()

f = open(filename)
```

```
analysis = {'blank_lines': 0,
            'cap_words': 0,
            'numbers': 0,
            'punctuation': 0,
            'same_start_end': 0}
```

Next, the program needs to read through the file. As you've seen, the best way to do that is with a `for` loop. Each iteration will return one line from the file, which can then be analyzed. The analysis from each line is added to the `analysis` dictionary, expanding it over time.

But what needs to be done with each line? There are five different measures that the exercise asked you to keep track of, and each requires doing something different with the string:

First, the program counts blank lines. It does this by invoking `one_line.strip()`, returning a version of `one_line` from which space and newline characters on the edges have been removed. If `one_line.strip()` returns the empty string, then the line was blank, and should be added to the count:

```
for one_line in f:
    if one_line.strip() == '':
        analysis['blank_lines'] += 1
```

You also needed to count the number of words in each of three categories. That required breaking the line into words (using `str.split`), and then iterating over each word in a nested `for` loop.

- To check if `one_word` is capitalized, you can invoke `one_word.capitalize()`, and compare the result with the original `one_word`. If they are the same, then the word is capitalized — that is, starts with a capital letter, with the rest being lowercase letters.
- To check if the first and last letters are the same, you can invoke `one_word.lower()` and then compare the first character, at index 0, with the final character, at index -1.

- To check if a word contains only digits 0-9, you can invoke `one_word.isdigit()`. If it returns `True`, then the word is a number.

Here is the code for that inner loop:

```
        for one_word in one_line.split():
            if one_word == one_word.capitalize():
                analysis['cap_words'] += 1

            if one_word.lower()[0] == one_word.lower()[-1]:
                analysis['same_start_end'] += 1

            if one_word.isdigit():
                analysis['numbers'] += 1
```

Finally, you needed to count the number of punctuation characters, defined as periods, question marks, and exclamation points. To find these, you need another nested loop — not over the words this time, but rather over each of the characters. Since `one_line` is a string, you can just invoke a `for` loop to go over it:

```
        for one_character in one_line:
            if one_character in '.?!':
                analysis['punctuation'] += 1
```

The program then finished by closing the file, followed by a `for` loop over `analysis.items()`, iterating over and printing each key-value pair:

```
    f.close()

    for key, value in analysis.items():
        print(f'{key}: {value}')
```

The full code looks like this:

```
    filename = input('Enter filename: ').strip()

    f = open(filename)
```

```
analysis = {'blank_lines': 0,
            'cap_words': 0,
            'numbers': 0,
            'punctuation': 0,
            'same_start_end': 0}

for one_line in f:
    if one_line.strip() == '':
        analysis['blank_lines'] += 1

    for one_word in one_line.split():
        if one_word == one_word.capitalize():
            analysis['cap_words'] += 1

        if one_word.lower()[0] == one_word.lower()[-1]:
            analysis['same_start_end'] += 1

        if one_word.isdigit():
            analysis['numbers'] += 1

    for one_character in one_line:
        if one_character in '.?!':
            analysis['punctuation'] += 1

f.close()

for key, value in analysis.items():
    print(f'{key}: {value}')
```

AI Challenge: Add a Bug

This time, I want you to present this program to a GenAI chatbot, and introduce a bug. Can the chatbot spot it? And can you predict how the bug will affect the program's outcome?

Here is a sample session with Gemini:

```
I'm working through an intro Python book. I just did an exercise
that
has to analyze a text file (chosen by the user) in five different
ways. I'm giving you a version of the program that I wrote, but I
have
introduced a bug. Can you tell me what the bug is?

filename = input('Enter filename: ').strip()
```

```
f = open(filename)

analysis = {}

for one_line in f:
    if one_line.strip() == '':
        analysis['blank_lines'] += 1

    for one_word in one_line.split():
        if one_word == one_word.capitalize():
            analysis['cap_words'] += 1

        if one_word.lower()[0] == one_word.lower()[-1]:
            analysis['same_start_end'] += 1

        if one_word.isdigit():
            analysis['numbers'] += 1

    for one_character in one_line:
        if one_character in '.?!':
            analysis['punctuation'] += 1

f.close()

for key, value in analysis.items():
    print(f'{key}: {value}')
```

Gemini responded:

```
It's great that you're working through these exercises! Analyzing
text
is a classic way to get comfortable with loops and dictionaries.

The bug in your code is a **`KeyError`**.

### Why the error occurs

In Python, you cannot update a value for a key that doesn't exist
yet
in a dictionary. When your code reaches a line like
`analysis['blank_lines'] += 1`, Python tries to do two things:

1. Look up the current value of `analysis['blank_lines']`.
2. Add `1` to it.
```

Since your dictionary (`analysis = {}`) starts out completely
empty,
Python fails at step 1 because the key `'blank_lines'` hasn't
been
created yet.

How to fix it

There are two common ways to solve this.

Option 1: Initialize the dictionary

Before you start your loop, define the keys you plan to use and
set
their starting values to `0`.

```python
analysis = {
    'blank_lines': 0,
    'cap_words': 0,
    'same_start_end': 0,
    'numbers': 0,
    'punctuation': 0
}

```

Option 2: Use `defaultdict` (The "Pro" Way)

Python has a built-in tool called `defaultdict` in the
`collections`
module. It automatically creates a key with a default value (like
`0`)
the first time you try to access it.

```python
from collections import defaultdict

# This tells Python: "If a key is missing, assume its value is 0"
analysis = defaultdict(int)

```

A few "Bonus" tips for your script

```
Since you're learning, here are two small improvements to make
your code more robust:

1. **Use the `with` statement:** It is safer to open files using
`with
open(filename) as f:`. This ensures the file closes automatically
even
if the program crashes, so you don't need `f.close()`.

2. **The `.capitalize()` logic:** Be careful with `one_word ==
one_word.capitalize()`. The `.capitalize()` method forces the
first
letter to uppercase **and all other letters to lowercase**. If
your
text has a word like "NASA", `.capitalize()` turns it into
"Nasa", and
your check will return `False`. If you just want to see if it
starts
with an uppercase letter, use `one_word[0].isupper()` instead.

**Would you like me to show you how the final, optimized version
of
  this script would look with those improvements?**
```

Writing to Files

By default, opening a file in Python puts it into read-only mode. That is, you can read from the file, but you cannot write to it. Since you'll likely read from files more often than you'll write to them, this is quite reasonable.

But what if you do want to write to a file? If files provide a way to store data structures, then there must be a way to convert those data structures to a file, not just the other way around (which you've been doing so far).

Writing to a file is similar to reading from one. There are two main differences. First, you must specify that the file is to be opened in "write" mode. Typically this is done by passing `'w'` as the second argument to `open`. Note that `'w'` is potentially quite dangerous, because it removes any content that the file might already contain. (More on this in "Reading and Writing".)

Second, you write to the file by invoking the `write` method. This method takes a string argument, which is written to the file verbatim. Unlike `print`, `write` doesn't add a newline character after writing to a file — so if you want to end the line, you must do so explicitly.

Here is an example of writing to a file:

```
f = open('output.txt', 'w')

f.write('Hello\n')
f.write('Hello again!\n')
f.write('I have nothing more to say. Goodbye!\n')

f.close()
```

After running the preceding program, you'll find a text file in the same directory as the program was run, which looks like:

```
Hello
Hello again!
I have nothing more to say. Goodbye!
```

Flushing and Closing

The preceding code works, but it contains several potential issues — issues that are so common among Python developers that there are standard solutions for them.

The first has to do with how data is written to a file: if every invocation of `write` would really write to the disk, then your computer would slow down dramatically. That's because even the fastest disks are far slower than a computer. Writing to the disk each time a program demands it would force the computer to spend time waiting for the disk to finish.

The solution, used by most modern operating systems, is to *buffer* the output. That is, every time your program asks the operating system to write something to the filesystem, the OS actually writes it to a block of memory. Only when the buffer fills up is the data actually written to the disk. By

batching the writes, the computer reduces the amount of time it has to spend waiting for the disk to finish writing.

Normally, buffered output is a good thing, and is fairly invisible to the programmer and the user. However, if your program only writes a small amount of output, then it might remain in the buffer for quite some time. You can force the buffer to be flushed, or emptied, by invoking the `flush` method on the file object, as in `f.flush()`. Closing the file with `f.close()` flushes the buffer before closing the connection, too.

This is such a common issue that Python provides a standard syntax for working with it, the `with` statement. `with` automatically flushes and closes a file, meaning that you don't need to think about it. The only tricky part is the `with` statement itself, which turns the usual Python assignment syntax on its head. Rather than saying `f = open(filename)`, you say `with open(filename) as f:`.

The `:` at the end of the line means that `with`, like `if` and `for`, then starts an indented block of code. Inside the `with` block, the file is open, which means that you can write to it. Once the `with` block ends, though, it is flushed and closed, meaning that you cannot write to it anymore.

Here is the preceding program, rewritten to use `with`:

```
with open('output.txt', 'w') as f:
    f.write('Hello\n')
    f.write('Hello again!\n')
    f.write('I have nothing more to say. Goodbye!\n')
```

`with` can be used in a number of Python contexts, but in my experience it is mostly used with files. Over your Python career, though, you might see it in other places. The basic idea is that `with` turns to the value to its right — the file object, in this case — and asks what it wants to do at the start of the indented block, and again at the end of the indented block. If the value wants to do something special at either of those points in time, then `with` follows those instructions.

In other words, you can think of the preceding code in the following way:

```
with open('output.txt', 'w') as f:

    # with asks f: anything special to do at the block start?
    # f responds: no, keep going

    f.write('Hello\n')
    f.write('Hello again!\n')
    f.write('I have nothing more to say. Goodbye!\n')

    # with asks f: anything special to do at the block end?
    # f responds: yes, flush + close me
```

Could you also use `with` when reading from files? Yes, absolutely! Failing to close a file doesn't have as many problems when you're just reading from a file. But it is considered a best practice, as well as tidier and more standard, to use `with` when reading from a file, too. Here is a program that you saw earlier in this chapter, modified to use `with` and annotated with comments reflecting what's going on:

```
total = 0
filename = input('Enter a filename: ').strip()

with open(filename) as f:

    # with asks f: anything special to do at the block start?
    # f responds: no, keep going

    for one_line in f:
        total += len(one_line)

    # with asks f: anything special to do at the block end?
    # f responds: yes, flush + close me

print(f'Total size: {total}')
```

`with` works well when all of your file-related operations are located close to one another in your program. If you need to read from or write to a file in more than one part of your program, then you might not be able to use `with`. In such a case, stick with the traditional `close` method.

Reading and Writing

You have seen, so far, that you can open a file in read-only mode (`'r'`) or in write-only mode (`'w'`). What if you want to both read from and write to a file at the same time?

In theory, you can do that, using the special `'r+'` option to `open`. However, I recommend that you avoid doing this. Doing both operations on the same file can confuse your reading, your writing, or both. It's not as simple as inserting text into a document in your word processor. Both reading and writing move the computer's "current location" in the file, which means that reading affects where you're writing, and vice versa. Moreover, writing to the middle of a file replaces existing text, rather than pushing it to the right.

Opening a file for both reading and writing at the same time can thus lead to a confusing mess. That's bad enough when the file contains text, but can be truly catastrophic if the file is formatted in a specific way, to be read by other programs.

There are, however, two options to `open` that you might find useful:

- The `a` option opens a file for appending, meaning that writing to the file adds text to the end, after any existing text. This is perfect for logfiles, where you want to keep anything that was in the file before, and also put new information at the end.
- The `x` option opens a file for writing, but exits with an error if the file already exists. This avoids the all-too-common situation in which you open a file for writing with `'w'`, which destroys any existing content, then realize you made a mistake…but the previous content is already gone. Opening a file with `'x'` avoids this problem, ensuring that you only write to brand-new files.

Exercise: Dict to Config File

Earlier, you saw how to read data from a text file and turn it into an in-memory data structure. In this exercise, you'll take a dict in memory and

write it to a text file on disk.

Specifications

Here is what you need to implement for this exercise:

1. Define a dictionary in which the keys and values are a combination of strings and integers.
2. Using `with`, write the contents of the dict to a text file. Each line of the file should contain one key-value pair, separated by =, as in `a=10`.
3. When you're done, look at the file and check that it matches the dictionary.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

Before doing anything else, the program needs to define a dict:

```
d = {'a':10, 'b':20, 'c':30}
```

Next, the program opens the file for writing, using `with`, assigning the file object to `f`:

```
d = {'a':10, 'b':20, 'c':30}

with open('pairs.txt', 'w') as f:
```

Inside the `with` block, you'll need to iterate over `d`, one pair at a time, using a `for` loop:

```
d = {'a':10, 'b':20, 'c':30}

with open('pairs.txt', 'w') as f:
    for key, value in d.items():
```

Finally, you can write the current key-value pair to the file with `f.write`, passing it a string. In this case, it's probably best to use an f-string with `key` and `value`. Remember that you need = between `key` and `value`, and that you need to add the `'\n'` at the end of each string:

```
d = {'a':10, 'b':20, 'c':30}

with open('pairs.txt', 'w') as f:
    for key, value in d.items():
        f.write(f'{key}={value}\n')
```

AI Challenge: Specify and Compare

Give a GenAI chatbot the same specifications that you got for this exercise. Ask it to solve the problem. Compare your solution and the one it gives you with the one here in the book. What similarities and differences do you see?

Here's how such a session went with ChatGPT:

```
I'm doing an exercise in an intro Python class. I've solved the
exercise already, but want to see how you would solve it, given
the
same specifications. Then I'll compare your solution with mine.

Exercise: Dict to config file

Earlier, you saw how to read data from a text file and turn it
into an
in-memory data structure. In this exercise, you'll take a dict in
memory and write it to a text file on disk.

* Define a dictionary in which the keys and values are a
combination
  of strings and integers.
* Using `with`, write the contents of the dict to a text file.
Each
  line of the file should contain one key-value pair, separated
by
  `=`, as in `a=10`.
* When you're done, look at the file and check that it matches
the
  dictionary.
```

ChatGPT responded:

````
Here's a straightforward way I'd solve it in "intro Python" style
— clear,
minimal, and matching the specs.

```python
# 1) Define a dictionary with string/int keys and values
config = {
    "host": "localhost",
    "port": 5432,
    10: "ten",
    3: 99
}

# 2) Write it to a text file, one key=value pair per line
filename = "config.txt"

with open(filename, "w", encoding="utf-8") as f:
    for key, value in config.items():
        f.write(f"{key}={value}\n")

# 3) (Optional) Read it back just to visually verify
with open(filename, "r", encoding="utf-8") as f:
    print(f.read())
```

Notes (still within the exercise vibe):

* `f"{key}={value}"` automatically converts ints to strings when
  formatting.
* The order of lines will match the dict's insertion order in
modern
  Python, but the exercise usually doesn't require a specific
order.
````

Summary

In this chapter, you learned how to read from and write to files — skills that allow you to save data from a computer's in-memory data structures to a file, and then to read from that file back into in-memory data structures. Along the way, you learned about opening and closing files, iterating over a file with `for`, and about automatically flushing and closing files using

`with`. With basic data structures and files under your belt, you're ready to advance to the next chapter, and with it to the next level of programming: writing your own functions.

Chapter 9. Functions

So far, this book has concentrated on Python's nouns, the data structures in which you store information. You have seen how Python's built-in data structures, such as strings, lists, tuples, and dicts, are both flexible and powerful, reflecting decades of real-world use and improvements. You've even seen how you can create custom combinations of these data structures that more closely reflect the real world.

But important as data structures might be, programming is often about verbs — functions — rather than nouns. As you've already seen, you can do a lot with Python's built-in functions and methods, including `print`, `input`, and `str.strip`. But the real magic happens when you write your own functions.

Part of this is the DRY rule ("don't repeat yourself") that you first learned about in Chapter 5, in the context of loops. If your program repeats the same actions on a regular basis, you can bundle those actions into a function. That makes your program clearer and easier to read, and also ensures that when (not if!) you have to fix or change your code, you'll only have to do it in one place.

But functions also allow you to think about your program at a higher level of abstraction, ignoring the low-level details as you build larger programs. You do this in real life, as well: if someone asks you what you had for breakfast, and you had an omelette, you'll just say, "I made an omelette."

You're unlikely to say, "I put a pan on the stove, and turned on the gas. I got some eggs from the refrigerator. I put oil in the pan. I cracked eggs into a bowl, added some milk, and whisked it all together. Then, after the pan was sufficiently hot, I added the eggs. When the eggs were largely solid, I used a spatula to flip them over. When it was evenly cooked, I took the omelette out of the pan, and turned off the gas." This longer explanation is no less

accurate, but goes into unnecessary detail, especially if someone already knows what it means to make an omelette.

Besides, what if you had four friends over, and made an omelette for everyone? A higher-level abstraction allows you to say, "I made five omelettes" rather than repeating your lengthy, detailed description.

You've already benefitted from the way in which functions hide lower-level details: you have invoked `print` many times so far in this book, without thinking about what is really going on behind the scenes. You *could* learn what `print` is doing, since Python is open-source software, meaning that its code is open to all. But you're unlikely to do that. By writing functions, you allow yourself and other developers to think at a higher level, concentrating on how to solve the problems at hand, and not thinking about the underlying implementation.

Also, software can become quite complex, and often requires that you keep track of many different things. Dividing a program into functions means that you only have to remember what the functions do, rather than how they do those things, which is easier to understand and remember.

In this chapter, you'll learn how to define new functions (i.e., verbs) in Python. You'll also learn what happens to the values (*arguments*) that you pass to a function when you invoke it — namely, how they are passed to a function's *parameters*. You'll also learn about how to return a value from a function. By the end of this chapter, you'll know how to write functions, and will even have experience writing a few of your own.

Writing a Function

Defining a function in Python is similar to writing a recipe down for a friend: each of the individual steps is familiar, but the sum total of those steps produces something new and different. After you have defined a function, the language knows a new verb, one that you can use whenever you want in your program.

A function definition follows a straightforward pattern:

- The `def` keyword, indicating that you're starting to define a new function.
- The function's name. The same conventions that you've learned for variable names also apply to function names: all lowercase letters and digits, using _ between words.
- After the function's name, you put `()` with parameter names between them. You'll learn about parameters later in this chapter; for now, just leave those parentheses empty.
- At the end of the line, put a `:` — just like you've seen before, with `if`, `for`, and `while`.
- Then, starting on the next line, comes the "function body," an indented block of code. The function body must contain at least one line, but continues for as long as you want, ending only when the indented block ends.

For example, here is a simple function definition:

```
def hello():
    print('Hello!')
```

Once Python executes the preceding two lines, the `hello` function is now a part of its vocabulary, no more and no less than `print` and `input`. You can thus invoke it by naming the function and putting `()` after its name:

```
hello()
```

Running the preceding code results in the following being printed on the screen:

```
Hello!
```

There's a big difference between *defining* a function, which you normally do once, and *invoking* the function (also known as "calling" the function),

which you can do any number of times. To define the function, you'll use `def`, specifying the steps that you want the function to take. But when you call the function with `()`, that tells Python to execute the function body.

This is an important point, so it bears repeating: without the `()` following the function's name, it will not execute. `hello`, by itself, describes a function, but doesn't run it. `hello()`, by contrast, does execute, printing `Hello!` on the screen. If you forget `()` from a function call, then your program certainly won't do what you want — although it might not produce any error message. In some cases, though, it will lead to an error message, often one that is hard to debug.

The function body can be of any length, and can contain any Python code you want — from `input` and `print` to conditions with `if`, to loops with `for` and `while`. A function can also invoke one or more other functions. That's a great way to avoid having the function body get too long, thus making it hard to remember, understand, and maintain.

If a function contains an indented block, such as `if` or `for`, then it'll contain two or more levels of indentation. For example:

```
def hello():
    name = input('Enter your name: ').strip()

    if name == 'Reuven':
        print('Hello, boss!')

    else:
        print(f'Hello, {name}! Here is your name, spelled out:')
        for one_character in name:
            print(one_character)
```

The preceding code contains several blocks:

- The function body is the first indented block, coming right after the `def hello` line.
- The `if` statement has an indented block, executed if the user's name is `'Reuven'`.

- The `else` statement has an indented block, executed if the user's name is *not* `'Reuven'`.
- Inside the `else` block is a `for` loop, which is then further indented.

Because `def` defines the function — that is, adds a function to Python's vocabulary — any invocation of the function must come *after* the `def` line. Trying to invoke the function before it is defined will result in a `NameError` exception. This is a surprise to people coming from some other programming languages, in which a function's definition and invocation can come in either order.

> **NOTE**
>
> The body of a function executes when you invoke the function, not when the function is defined.

In many programming languages, variables and function names are stored separately. That allows you to simultaneously have both a variable `x` and a function `x`. This is *not* the case in Python, however: function names are variables, just referring to a function value (i.e., a verb) rather than a data structure (i.e., a noun). In a given Python program, you can have a variable `x` or a function `x`, but not both, at least not at the same time. If you do somehow have both in the same program — a bad idea! — then the most recent definition will be active.

Exercise: Calculator

In this exercise, you will define a simple `calc` function that will ask the user to enter two numbers and an operator. The function will display the resulting math expression, as well as the solution, on the screen. This is the first of three versions of the calculator function you will write in this chapter.

Specifications

Here are your instructions for this exercise:

- Define a function, `calc`.
- When the function is invoked, it will ask the user to do three things:
  - To enter a first number
  - To enter an operator, which should be either + or −
  - To enter a second number
- If either input isn't a number, print an error message on the screen.
- If the operator is neither + nor −, then the result should be `(unknown)` or something similar.
- Print the original numbers and operator, along with the result, on the screen.
- Execute the function, and check that it works.

For example:

```
Enter first number: 10
Enter operator: +
Enter second number: 5

10 + 5 = 15


Enter first number: 8
Enter operator: *
Enter second number: 3

8 * 3 = (unknown)


Enter first number: hello
Enter operator: +
Enter second number: goodbye
```

```
Your input for first and/or second was not numeric.
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

In order to define a function named `calc`, you need to start with `def calc`:

```
def calc():
```

The rest of the solution takes place inside `calc`, in the function body. And the first part of that consists of three calls to `input` and assignments to three variables: `first`, `operator`, and `second`:

```
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()
```

Next, you need to check if `first` and `second` can be turned into integers. You can do that with a combination of `if` and the `str.isdigit` method, combined with `and`. If both return `True`, then you can invoke `int` on both `first` and `second`, getting back new integer values, assigned here to `n1` and `n2`:

```
def calc():
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()

    if first.isdigit() and second.isdigit():
        n1 = int(first)
        n2 = int(second)
```

Next, the program assigns a value to `result`, depending on the contents of `operator`:

```
def calc():
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()

    if first.isdigit() and second.isdigit():
        n1 = int(first)
        n2 = int(second)

        if operator == '+':
            result = n1 + n2
        elif operator == '-':
            result = n1 - n2
        else:
            result = '(unknown)'
```

With `result` defined, the function can then print the full expression for the user, using an f-string:

```
def calc():
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()

    if first.isdigit() and second.isdigit():
        n1 = int(first)
        n2 = int(second)

        if operator == '+':
            result = n1 + n2
        elif operator == '-':
            result = n1 - n2
        else:
            result = '(unknown)'

        print(f'{first} {operator} {second} = {result}')
```

Finally, if either `first` or `second` returned `False` from `str.isdigit`, the `else` block fires, scolding the user about their inputs:

```
def calc():
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()
```

```
    if first.isdigit() and second.isdigit():
        n1 = int(first)
        n2 = int(second)

        if operator == '+':
            result = n1 + n2
        elif operator == '-':
            result = n1 - n2
        else:
            result = '(unknown)'

        print(f'{first} {operator} {second} = {result}')

    else:
        print(f'Your input for first and/or second was not
numeric.')
```

Success! This function turns your expensive computer into a cheap, underpowered pocket calculator.

GenAI Task: Find the Bug

This time around, you're going to ask GenAI to insert a bug into the program. Your job is to identify the bug, and what problems it is causing.

Here is my input to Gemini:

```
I'm learning Python functions, and my book has had me write the
following simple function:

def calc():
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()

    if first.isdigit() and second.isdigit():
        n1 = int(first)
        n2 = int(second)

        if operator == '+':
            result = n1 + n2
        elif operator == '-':
            result = n1 - n2
        else:
```

```
            result = '(unknown)'

        print(f'{first} {operator} {second} = {result}')

    else:
        print(f'Your input for first and/or second was not
numeric.')

I want you to insert a bug into this function. My job will then
be to
identify it, and tell you how it changes (breaks) things.
```

Here is Gemini's response:

```
That's a fantastic way to learn! Debugging is a core skill; in
fact,
professional developers often spend more time finding and fixing
bugs
than writing new code.

I have modified your function slightly to include **one
intentional
logic bug**. It will still "run" without crashing immediately,
but it
won't behave the way you expect under certain conditions.

### The Buggy Code

```python
def calc():
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()

    # I've made a change in the line below...
    if first.isdigit() or second.isdigit():
        n1 = int(first)
        n2 = int(second)

        if operator == '+':
            result = n1 + n2
        elif operator == '-':
            result = n1 - n2
        else:
            result = '(unknown)'
```
```

```
        print(f'{first} {operator} {second} = {result}')

    else:
        print(f'Your input for first and/or second was not
numeric.')

```
---

**Your Challenge:**

1. **Identify the bug:** Which specific keyword or symbol did I
change?

2. **The "Break" scenario:** If you run this code, what specific
input
could a user provide that would cause the program to **crash**
(throw
an error) instead of showing the "not numeric" message?

3. **The Logic:** Why does that specific change cause the crash?

**Would you like me to give you a hint, or are you ready to try
to spot it?**
```

My response:

```
Very sneaky--you used `or` instead of `and`, which means that
Python
will try to turn `first` and `second` into integers if even one
of
them contains only digits. Which will result in a crash when
invoking
`int` on the non-numeric string.
```

Arguments and Parameters

Consider the simple `hello` function from the start of the chapter:

```
def hello():
    print('Hello!')
```
```

There's nothing wrong with this function, except that it's a bit…boring. After all, you might want to let the user pass their name to the function as an argument, and then produce a custom message. You have passed arguments to functions before, and it stands to reason that you know what to do here, too:

```
hello('world')
```

However, in this particular case, you'll see the following error:

```
TypeError: hello() takes 0 positional arguments but 1 was given
```

In other words: you passed an argument to `hello`, but the function was defined not to accept any arguments. As such, instead of executing, the function raises an exception. That's because a function needs to know into which variable it can store each argument value. Such variables are known as *parameters*, and are declared on the first line of a function, in the parentheses just after the function's name.

For example, here is a version of `hello` that does accept an argument, storing it in the `name` parameter:

```
def hello(name):
    print(f'Hello, {name}!')
```

You can say that `hello` now accepts one argument, which will be assigned to the `name` variable — or more precisely, to the `name` parameter. And indeed, with that new definition in place, you can say:

```
hello('world')
```

Sure enough, the program prints out:

```
Hello, world!
```

By the way, the previous version of `hello`, which didn't take any arguments, is no longer available. That's because you have now defined `hello` to be a function that takes one argument. While some programming languages allow for more than one version of a function to exist simultaneously, and then invoke the version that best matches the arguments that were passed, Python is not one of them: you can only have one version of a function at a time. Defining `hello` with one parameter removed the zero-parameter version from Python's memory. As such, trying to invoke `hello` without any arguments won't work:

```
hello()
```

Invoking the preceding code results in this output:

```
TypeError: hello() missing 1 required positional argument: 'name'
```

In other words, the number of arguments and the number of parameters must match precisely. Any difference between the two will result in an exception. (In Chapter 10, you'll learn how to write functions that are more flexible in how many arguments they accept.)

For a function to accept two arguments, just put two parameters inside the `()`, separated by a comma:

```
def hello(first, last):
    print(f'Hello, {first} {last}!')
```

This version of `hello` can be invoked with two arguments:

```
hello('Reuven', 'Lerner')
```

The output:

```
Hello, Reuven Lerner!
```

In this invocation, the first argument (`'Reuven'`) was assigned to the first parameter (`first`), while the second argument (`'Lerner'`) was assigned to the second parameter (`last`). This is the most common way to call a function, and uses what are known as *positional arguments*. As the name indicates, positional arguments are assigned to parameters based on their position.

Dynamic Typing and Parameters

Consider the following function:

```
def add(x, y):
    total = x + y
    print(total)
```

As you might expect, you can invoke `add`:

```
add(10, 5)
```

You'll get the following result when you do:

```
15
```

But notice that nowhere does the function indicate what types of values must be passed as arguments. You can thus pass *any* values that respect the + operator:

```
add('abcd', 'efgh')
```

The preceding code results in this printout:

```
abcdefgh
```

You can even pass lists:

```
add([10, 20, 30], [40, 50, 60])
```

This invocation prints:

```
[10, 20, 30, 40, 50, 60]
```

This behavior is a reminder that Python is a *dynamically typed* language. Values have types (e.g., integer, string, or list), but variables do not. A variable — including a parameter — can refer to any type of value. This gives Python coders a great deal of flexibility, something that is normally seen as an elegant advantage over other languages. However, it also means that the following code won't raise any alarms until the program is run:

```
add('abcd', 20)
```

Only when the function runs, and tries to add `'abcd'` (a string) with `20` (an integer) do things blow up.

Exercise: Calculator with Arguments

In the previous exercise, you wrote a `calc` function. That function, once invoked, asked the user to enter two numbers and an operator, and used them in the calculation. In real life, however, it's more common to simply pass such values as arguments to a function.

For this exercise, you'll update the `calc` function such that it takes three arguments — a first number, an operator, and a second number — instead of asking the user to enter them each time the function is invoked.

Specifications

Here are your instructions for this exercise:

- Rewrite `calc`, such that instead of using `input` in the function body to get values for `first`, `operator`, and `second`, get them as arguments.
- You can assume that the caller is passing the right types of values — integers for `first` and `second`, and a string for `operator`.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

In this exercise, you evolved the original `calc` function to something more usable and modern. It's annoying, and less useful, for a function to ask the user for input. It also makes it harder to run automatic tests on the function, or to integrate it elsewhere in the program.

The goal of this exercise was to understand that whether input comes from the user via `input`, or via an argument, the value will still be stored in a variable. A parameter is a special variable, in that it is guaranteed to be assigned a value from whoever calls the program, via an argument.

Here is the original version of `calc`:

```
def calc():
    first = input('Enter first number: ').strip()
    operator = input('Enter operator: ').strip()
    second = input('Enter second number: ').strip()

    if first.isdigit() and second.isdigit():
        n1 = int(first)
        n2 = int(second)

        if operator == '+':
            result = n1 + n2
        elif operator == '-':
            result = n1 - n2
        else:
            result = '(unknown)'

        print(f'{first} {operator} {second} = {result}')

    else:
        print(f'Your input for first and/or second was not
numeric.')
```

If the function will get `first`, `operator`, and `second` all passed as arguments, then the calls to `input` can go away, and can be replaced by parameters with the same names:

```
def calc(first, operator, second):
    if first.isdigit() and second.isdigit():
        n1 = int(first)
        n2 = int(second)

        if operator == '+':
            result = n1 + n2
        elif operator == '-':
            result = n1 - n2
        else:
            result = '(unknown)'

        print(f'{first} {operator} {second} = {result}')

    else:
        print(f'Your input for first and/or second was not
numeric.')
```

But wait: the specifications indicated that `first` and `second` are guaranteed to be integers. In that case, the calls to `first.isdigit()` and `second.isdigit()` are not only unnecessary, they won't work — since `str.isdigit` cannot be invoked on integers. This means that both the `if` condition and its corresponding `else` can be removed:

```
def calc(first, operator, second):
    n1 = int(first)
    n2 = int(second)

    if operator == '+':
        result = n1 + n2
    elif operator == '-':
        result = n1 - n2
    else:
        result = '(unknown)'

    print(f'{first} {operator} {second} = {result}')
```

The calls to `int(first)` and `int(second)`, while they do no harm, are also unnecessary, assuming that both `first` and `second` are already integers. The function can thus shrink even more, using `first` and `second` directly, with no need for either `n1` or `n2`:

```
def calc(first, operator, second):
    if operator == '+':
        result = first + second
    elif operator == '-':
        result = first - second
    else:
        result = '(unknown)'

    print(f'{first} {operator} {second} = {result}')
```

This version of `calc` is shorter, easier to read and understand, and can integrate more easily into other programs. (And more improvements are on the way, in the next section.)

GenAI Task: Add a Bug

In the previous exercise, you asked AI to add a bug to your program, and you had to find it. This time, you'll be in the driver's seat, adding a bug and asking the GenAI system to find it. The point isn't to crash your program (although that can be fun!), but rather to understand what problems can arise even from subtle changes in the code. AI will hopefully not only find the problem, but also explain why it is an issue and how to avoid it.

I started by writing to ChatGPT:

```
I'm going through an introductory Python book, and am currently
on the
chapter about functions. I just wrote a simple function that lets
the
user pass two numbers (integers) and an operator (string, `+` or
`-`).

I'm going to give you the code that I wrote, which works.
However, I'm
going to add one bug. Can you find it?

def calc(first, operator, second):
    if operator == '+':
        result = first + second
    elif operator == '-':
        result = first - second
    else:
        result = '(unknown)'
```

```
    print('{first} {operator} {second} = {result}')
```

ChatGPT wrote:

````
The bug is in your `print` line: you're using `{...}`
placeholders,
but you didn't make it an f-string (or call `.format`), so it
will
literally print the braces and words.

Fix it with an f-string:

```python
print(f'{first} {operator} {second} = {result}')
```

Or with `.format`:

```python
print('{first} {operator} {second} = {result}'.format(
    first=first, operator=operator, second=second, result=result
))
```
````

Return Values

In many ways, the point of a function is to get a value back:

- When you invoke `len(s)`, you get the length of the value in `s`.
- When you invoke `type(s)`, you get the type of value in `s`.
- When you invoke `input()`, you get a string based on what the user typed.
- When you invoke `s.split()`, you get a list of strings based on the contents of `s`.

Programmers say that a function *returns* a value. That returned value can be assigned to a variable, or can be used as an argument to another function:

- `x = len(s)`
- `print(type(s))`
- `name = input('Enter your name: ')`
- `words = s.split()`

In order for a function to return a value, it must use the `return` statement. Note that `return` isn't a function, which means that the value you want to return doesn't need to be put inside parentheses. A function can return any type of value it wants, including a string, list, tuple, or dictionary. This allows a function to return complex values, or even multiple values as elements inside a list or tuple.

For example, here is a version of the `hello` program from earlier in this chapter, returning a string rather than printing it on the screen:

```
def hello(name):
    return f'Hello, {name}!'
```

You still execute the function in the same way:

```
hello('world')
```

But if you run that one line in a regular Python program, you won't see anything on the screen. Without an explicit call to `print`, a function's return value is not displayed anywhere. You can remedy that by passing the return value from `hello` to `print`:

```
print(hello('world'))
```

In the preceding code, `hello` returns a string value. That string value is, in turn, displayed on the screen by `print`.

If you are using Jupyter (as described in Chapter 1), then this distinction between returning and printing might be a bit confusing. That's because Jupyter, in trying to be helpful, automatically displays the value in the last

line of a cell — including the return value from a function. But don't be fooled! There is a huge difference between a function's return value and anything it might print on the screen.

A function's execution stops as soon as it reaches a `return` statement. However, a function can have multiple `return` statements. This is often the case inside an `if` statement, where each possible condition returns a different value to the caller:

```
def high_or_low(n):
    if n > 10:
        return 'high'
    else:
        return 'low'
```

The `high_or_low` function in the preceding code shows how a function can have more than one `return` statement, while knowing that only one will actually be executed on a given invocation.

What If a Function Doesn't Return Anything?

Some programming languages distinguish between functions, which return values, and *procedures*, which execute code but don't return anything. For example, you can imagine a procedure that erases a file; it's not obvious what value you would want to get back in such a case, so why bother?

Python doesn't make this distinction. Everything is a function, including methods, in that they must return a value.

But what if a function doesn't return a value? What if the function executes, gets to the end, and never encounters a `return` statement?

In such cases, a function returns a special value, known as `None`, mentioned in Chapter 6 in the context of the `list.append` method. It's not a string, or an integer, or even `False` — `None` is its own value, distinct from everything else, and is Python's way of saying that it needs to return *something*, but that in this case, something is actually nothing.

TYPE HINTS

This tension between the elegant flexibility of a dynamic language and the safety of a statically typed language has existed for many decades, and not just in the Python world. Now that Python is used in large, complex applications for large companies, where a great deal of money and safety are at stake, many developers are using *type hints* (also known as *type annotations*) to make it harder to pass the wrong kind of argument value to a function.

However, much to the surprise of many newcomers to Python, type hints are ignored by the Python language. Their enforcement requires the use of additional, external tools, such as `mypy` and `ty`.

To keep things as simple as possible, this book will ignore type hints. However, you should expect that you will, at some point, likely want to add them to your code.

Here is an example of how the `add` function would work with type hints, assuming that its usage should only be allowed for integers:

```
def add(x: int, y: int) -> int:
    total : int = x + y
    return total
```

And here is how the `high_or_low` function would look with type hints:

```
def high_or_low(n: int) -> str:
    if n > 10:
        return 'high'
    else:
        return 'low'
```

With these type hints in place, development tools will notice any invocations of `add` that don't pass integer values, and will flag them. But it's important to remember that the Python language itself doesn't enforce them.

Exercise: Calculator with Return Values

For the final exercise in this chapter, you'll again modify `calc`. This time, you'll change it to return a string value, rather than print that value on the screen. You'll then invoke the function a number of times inside a `for` loop.

Specifications

Here are your instructions for this exercise:

- Modify the `calc` program one final time, such that it returns a string value rather than printing it.
- Now use a `for` loop to invoke `calc` 10 times, each time with different numbers and operators.
- Print the result of each invocation on the screen.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The function itself will only have to change slightly, replacing the `print` invocation on the final line with a `return` statement:

```
def calc(first, operator, second):
    if operator == '+':
        result = first + second
    elif operator == '-':
        result = first - second
    else:
        result = '(unknown)'

    return f'{first} {operator} {second} = {result}'
```

The second part of the exercise asked you to invoke `calc` 10 times, each with different numbers. One easy way to do that is with `range`:

```
for number in range(5):
    calc(number, '+', number + 10)
    calc(number, '-', number + 10)
```

But if you invoke the preceding code, you won't see anything on the screen. That's because `calc` is now returning a value, not printing it. You need to take the string that `calc` returned, and pass it to `print`:

```
for number in range(5):
    print(calc(number, '+', number + 10))
    print(calc(number, '-', number + 10))
```

If you prefer, you could instead take the result from `calc`, assign it to a variable, and then print that:

```
for number in range(5):
    result = calc(number, '+', number + 10)
    print(result)

    result = calc(number, '-', number + 10)
    print(result)
```

Do you see some repeated code here? The `calc` function and `print` are each invoked twice, with the only difference between the first invocation and the second being the operator, switching from + to −. A `for` loop can fix that up:

```
for number in range(5):
    for operator in '+-':
        result = calc(number, operator, number + 10)
        print(result)
```

In this newer version, the outer loop still iterates over `range(5)`. The inner loop then iterates over the two-character string `'+='`, invoking `calc` once for + and a second time for −.

The new code produces the following result:

```
0 + 10 = 10
0 - 10 = -10
1 + 11 = 12
1 - 11 = -10
2 + 12 = 14
2 - 12 = -10
3 + 13 = 16
3 - 13 = -10
4 + 14 = 18
4 - 14 = -10
```

GenAI Task: Interview

With this version of `calc` (and the `for` loop), you now have a good sense of how to define functions, including functions that take arguments and return values. You've also seen how you can treat your own functions as you previously did Python's built-in functions, invoking them with arguments and using their return values.

For the GenAI task this time, put the definition of `calc`, as well as the `for` loop, and ask AI to interview you about the code.

```
I'm reading an intro Python book, and am currently going through
the
chapter on functions. In the most recent exercise, I wrote a calc
function that takes numbers and an operator, and returns a string
with
the result.

Here is the function, and a loop I used to invoke it:

    def calc(first, operator, second):
        if operator == '+':
            result = first + second
        elif operator == '-':
            result = first - second
        else:
            result = '(unknown)'
        return f'{first} {operator} {second} = {result}'

    for number in range(5):
```

```
        for operator in '+-':
            result = calc(number, operator, number + 10)
            print(result)
```

```
Please interview me about this function and its invocation,
asking
me one question at a time about it. After I give each answer,
evaluate what I said, correct any mistakes, and ask again until I
tell you to stop.
```

Claude responded:

```
Great! Let's start with a fundamental question about your `calc`
function:

**What does your function return, and what type of data is it?**
```

I wrote:

```
It returns an integer, the result of the math problem.
```

Claude said:

```
Actually, that's not quite right. Let's look at your function
more carefully.

Your function has this line at the end:

```python
return f'{first} {operator} {second} = {result}'
```

Notice the `f'...'` - that's an f-string. So your function is
actually
returning a **string**, not an integer. The string contains the
entire
equation formatted nicely, like `"5 + 15 = 20"`, rather than just
the
number `20`.

The variable `result` inside your function *is* an integer (or
could
be the string `'(unknown)'`), but that's not what the function
returns. The function wraps that result inside a formatted string
```

```
before returning it.

Does that make sense?

**Next question: In your nested loops, why do you think the code
uses
`for operator in '+-'` instead of something like `for operator in
['+', '-']`?**
```

Summary

In this chapter, you learned how to teach Python new verbs by defining functions. You learned how to use `def`, how arguments are assigned to a function's parameters, and how a function returns values to its caller with the `return` statement. Programmers spend a great deal of their time writing functions, partly because they help to break a program down into smaller (and more easily designed) pieces, and partly so that you can invoke the function multiple times in a program. In the next chapter, you'll learn how to make your functions easier to read, write, use, and maintain, building on everything you've learned here.

Chapter 10. More Functions

Functions, as you learned in Chapter 9, are an important part of programming: they allow you to break problems down into smaller, more manageable parts. They allow you to reuse functionality. And they allow you to work at a higher level of abstraction, letting you build larger, more complex projects without getting bogged down in the details.

You've also learned how functions communicate with other parts of a program: their inputs are passed as arguments, which are then assigned to parameters. The number of arguments that a function can accept is normally equal to the number of parameters. Those arguments can be of any type — strings, lists, tuples, dicts, or even more complex types.

Given how fundamental functions are to programming, you probably won't be surprised that there's much more to writing functions than you've seen so far. In this chapter, you'll go deeper into some of these techniques, making it easier to write modular, readable, and maintainable code. You'll also continue to see how Python tries to provide its users with a consistent approach.

Function Documentation

Before you can invoke a function, you need to know how many arguments it expects to receive, and also what types of values it'll accept. For example, if you call `len('a', 'b')`, then Python will give you an error message, indicating that `len` only takes one argument. But if you call `len(123)`, you'll get an error indicating that `123`, an integer, doesn't have a length.

But how are you supposed to know? In the case of functions that come with Python, you can always check the documentation at *https://docs.python.org*. If you're writing your Python code within an editor such as VS Code or PyCharm, then you see documentation by hovering with the mouse cursor

over the function's name. You can even see documentation in Jupyter or a similar notebook by invoking the `help` function, passing the name of the function you want to learn about as an argument:

```
help(print)
```

What about your own functions? What happens if you, or a colleague working with you, wants to get documentation about some code that you have written? For example:

```
def hello(name):
    return f'Hello, {name}'
```

If you then say:

```
help(hello)    # note: No () here!
```

You won't see much, and certainly not much that's helpful:

```
Help on function hello in module __main__:

hello(name)
```

True, you do see that this `hello` function takes a single argument, one that will be assigned to the parameter `name`. But beyond that — what types of values the function expects to receive, what it does, and what it returns — you really don't know anything.

Function documentation in Python is written using *docstrings*, short for "documentation strings." If the first line of a function is a string, then that string is considered the function's docstring. For example:

```
def hello(name):
    'This is a very friendly function!'
    return f'Hello, {name}'
```

Notice that the docstring isn't assigned to a variable. It's just a string, sitting there on the first line of the function. It doesn't affect the execution of the function, either; Python notices the docstring when it defines the function, and stores it away as part of the function's definition. Then, when someone asks for help about the function, Python displays this string:

```
help(hello)    # get help on the redefined function
```

The docstring is displayed:

```
Help on function hello in module __main__:

hello(name)
    This is a very friendly function!
```

But of course, you'll want your docstrings to be a bit more interesting and useful than that. In particular, you'll want to tell anyone using the function:

- An overall description of what the function does
- What the function *expects* to receive — the number and types of arguments
- What the function *modifies* while it is executing — changes to data structures, or even to files
- What the function *returns* to the caller

Because the docstring is a string, you can put whatever text you want inside it. However, it's conventional for a docstring to contain multiple lines, thanks to the use of a triple-quoted string (i.e., one with """ and """ around it). The first line of the docstring is a general description of what the function does. You can then include sections for "Expects," "Modifies," and "Returns." You can then add one or more examples of its usage, if you like.

For example:

```
def hello(name):
    """Greet the user in a friendly way.
```

```
    Expects: One argument, a string -- the user's name
    Modifies: Nothing
    Returns: A new string, a greeting that includes the user's
name
    """
    return f'Hello, {name}'
```

Sure enough, invoking `help(hello)` after the preceding definition produces this output:

```
Help on function hello in module __main__:

hello(name)
    Greet the user in a friendly way.

    Expects: One argument, a string -- the user's name
    Modifies: Nothing
    Returns: A new string, a greeting that includes the user's
name
```

How Are docstrings Different from Comments?

At this point, you might be wondering: why write docstrings? I'm already writing comments in my code, starting with `#`.

And it's indeed a great thing that you're writing comments! But docstrings are different, both in content and intended audience.

Comments are meant for whoever will be *maintaining* the function after you have written it. (This might even include you.) They provide information about the intention of the program, as well as anything nonobvious that might surprise or cause trouble for someone who will change it in the future. You might, for example, use comments to explain a complex algorithm or data structure. Comments aren't meant to be seen by people invoking the function. They're aimed at people who will maintain the function in the future. They are similar to your car's maintenance

manual, which is meant for the garage that will service it, not the owners and drivers.

Docstrings, by contrast, are meant for someone *using* your function in their own program. They have to know what arguments to send, what they'll receive, and what the function will do while it's running. But they don't need to know the inner workings. Docstrings shouldn't mention algorithms or data structures, unless they're particularly important for the end user. A docstring is similar to your car's owner's manual, in that it tells you how to use the system, but not how it's implemented inside or how to fix it.

As a result, it's reasonable for a function to have comments (to help future maintainers) and also docstrings (to help future users), and for them not to overlap very much, if at all.

Exercise: Document the Calculator Function

In Chapter 9, you wrote a `calc` function that takes three arguments — two numbers and an operator — and returns a string indicating the result of performing that operation on those two numbers. Here, you'll add a docstring to that `calc` function.

Specifications

In this exercise, you are to add a docstring to `calc`, including several examples of its invocation.

You can solve this exercise with the help of an AI tutor.

Solution

Here is a version of the function with a full docstring. Notice several things about this docstring:

- It indicates each parameter's name, and what types of values it expects to receive.

- It indicates acceptable values for each argument.
- It says what happens if you pass bad values for "operator".
- It includes examples of all of these.

Notice that the docstring does *not* mention what happens if the user invokes `calc` with non-numeric values. That's because the assumption is that the caller will only use numeric values — passing a string as `first` or `second` would be like passing an integer to `len`, a violation of the function's docstring, and thus the function's assumptions.

```
def calc(first, operator, second):
    """Perform a math operation on two numbers, and get back a
    string with the full expression.

    Expects: Three arguments:
            - first, an integer or float
            - operator, a string, either '+' or '-'
            - second, another integer or float
    Modifies: Nothing
    Returns: A string containing a math expression based on the
        numbers and operator, including the result.

    If the operator is unknown, then the result is listed as
    "(unknown)" in the returned string.

    Examples:

        calc(10, '+', 3)  # returns '10 + 3 = 13'
        calc(10, '-', 8)  # returns '10 - 8 = 2'
        calc(10, '*', 3)  # returns '10 * 3 = (unknown)'
    """

    if operator == '+':
        result = first + second
    elif operator == '-':
        result = first - second
    else:
        result = '(unknown)'
    return f'{first} {operator} {second} = {result}'
```

AI Challenge: Specify and Compare

For this GenAI task, provide the function to a GenAI system. Ask it to write a docstring for `calc`. Compare the output docstring with the one that you wrote. What did GenAI include that you didn't? What did you include that it didn't?

Here is what I wrote to Claude:

```
I'm reading an intro Python book, and have been asked to write a
docstring for the following function:

    def calc(first, operator, second):
        if operator == '+':
            result = first + second
        elif operator == '-':
            result = first - second
        else:
            result = '(unknown)'
        return f'{first} {operator} {second} = {result}'

Please write such a docstring for the `calc` function.
```

Claude responds:

```
def calc(first, operator, second):

    """Perform a basic arithmetic operation and return a
formatted
    string showing the equation.

    Takes two numeric operands and a string operator, calculates
the
    result for addition and subtraction, and returns a human-
readable
    string in the form 'first operator second = result'. If the
    operator is not recognized, the result is shown as
'(unknown)'.

    Args:
        first: The left-hand operand (numeric).
        operator: A string representing the operation ('+' or '-
').
        second: The right-hand operand (numeric).

    Returns:
```

```
        A formatted string such as '3 + 5 = 8' or '10 * 2 =
        (unknown)'.
    """
```

Not very far off from what I had provided! Notice that the description went beyond a simple one-line headline, and gave a good human-readable description.

Default Argument Values

Consider this function, which you have already seen:

```
def add(x, y):
    return x + y
```

You can obviously invoke `add` with two arguments, as in `add(10, 3)`. What happens if you try to invoke it with fewer than two arguments? Or with more? Python will raise an error:

```
add(10)           # TypeError: add() missing 1 required
positional argument: 'y'
add(10, 20, 30)   # TypeError: add() takes 2 positional arguments
                  #            but 3 were given
```

This makes a lot of sense; you defined the function with two parameters, so someone invoking it must pass two arguments.

And yet, something here doesn't quite add up. You have already seen that in some cases, Python is a bit more forgiving. For example, the `str.split` method can take an argument, indicating what string should be used as a field separator. Or it can be invoked without any arguments, in which case whitespace is the separator:

```
s = 'ab:cd ef:gh'
s.split(':')    # Returns the 3-item list ['ab', 'cd ef', 'gh']
s.split()       # Returns the 2-item list ['ab:cd', 'ef:gh']
```

As you can see from the preceding code, it's possible to call `str.split` with no arguments (in which case it defaults to the use of whitespace) or with one argument (in which case it uses the provided string as a delimiter). How is this possible? More specifically, how can you write functions with this flexible behavior?

The trick is to provide a default argument value to a parameter when defining the function. In other words, in the first line of the function, when you mention the parameter, just put a = sign after its name, and the default value you want to provide. For example:

```
def add(x, y=10):
    return x + y
```

You can now invoke `add` with either one or two arguments:

```
add(12, 5)   # returns 17
add(12)      # returns 22
```

In the first case, each argument is assigned to a parameter:

- The first argument is `12`, so it is assigned to the first parameter, `x`.
- The second argument is `5`, so it is assigned to the second parameter, `y`.

Python calculates `x + y`, which is the same as `12 + 5` in this case, and returns the integer `17`.

The second case is a bit more complex:

- The first argument is 12, so it is assigned to the first parameter, `x`. This doesn't change.
- There is no second argument! But `add` requires it to give `y` a value.
- Python turns to the `add` function, and checks if a default argument value for `y` was set when `add` was defined.

- The default value `10` is found and assigned to `y`.

Now the function's body can execute — without any obvious knowledge or acknowledgment that `y` was assigned from a default, rather than from the caller's arguments. Python calculates `x + y`, which is the same as `12 + 10`, and returns the integer `22`.

You can set default argument values to more than one parameter:

```
def add(x=5, y=10):
    return x + y
```

You can now invoke `add` with either zero, one, or two arguments:

```
add(12, 5)   # returns 17
add(12)      # returns 22
add()        # returns 15
```

A default argument value doesn't *really* make a parameter optional, even though that's how we often think and talk about it. Rather, it means that the parameter will get its value from the stored default value that was set when the function was defined.

A function can have as many, or as few, default argument values as you like. However, all parameters with defaults must come *after* all parameters without defaults. Put a different way, mandatory parameters must come before optional parameters. For example, consider:

```
def add(x=5, y):     # first param x has a default, but y doesn't
    return x + y
```

Trying to define the preceding function will result in an error:

```
SyntaxError: parameter without a default follows parameter with a
default
```

When should you define a function with a default argument value? One rule of thumb is that if your function will normally be used in one way, but you

want to allow for other ways, then you can use a default. Some examples:

- As you already saw, `str.split` defaults to treating all whitespace characters, in any quantity, as a separator. This is done via a default value of `None` for the `sep` parameter.
- `str.split` has a second optional parameter, `maxsplits`, with a default value of -1. Providing an integer caps the number of elements the returned list of strings will contain.
- `open` defaults to opening a file for reading, via a default value of `'r'` for the second (`mode`) parameter.
- `int`, which returns an integer based on a string, normally treats the string as a decimal (base 10) number. That's because the second parameter, `base`, has a default value of `10`. Passing a different value for `base` changes the way in which the string is interpreted. For example, `int('1010', 2)` returns the integer `10`, because `'1010'` is 10 in binary (aka base 2).

Consider a function that takes an argument, a list of numbers, and sums them:

```
def mysum(numbers):
    total = 0

    for one_number in numbers:
        total += one_number

    return total
```

This function works just fine; invoking it as `mysum([10, 20, 30])` returns the int `60`.

Maybe, though, the people who will run this function sometimes want to sum numbers above a certain threshold. That is, they want to ignore any number that doesn't meet that threshold. Not everyone needs this

functionality, but it's common enough that it should be included in the function.

You could rewrite the function as follows:

```
def mysum(numbers, threshold):
    total = 0

    for one_number in numbers:
        if one_number >= threshold:
            total += one_number

    return total
```

This version of the program works just fine; if you invoke `mysum([10, 20, 30], 15)`, then it'll return 50, the sum of 20 and 30. The number 10 is ignored, because it isn't greater than or equal to 15, the value of `threshold`.

> **NOTE**
>
> This version of `mysum`, with the `threshold` parameter, assumes that all of the input numbers are positive.

If `threshold` is only going to be set on occasion, then it makes more sense to give it a default argument value, say of 0:

```
def mysum(numbers, threshold=0):
    total = 0

    for one_number in numbers:
        if one_number >= threshold:
            total += one_number

    return total
```

This version of `mysum` defaults to having a threshold of 0, but that can be set or changed by the caller:

```
mysum([10, 20, 30])         # returns 60, since threshold=0
mysum([10, 20, 30], 15)     # returns 50, since threshold=15
mysum([10, 20, 30], 99)     # returns 0, since threshold=99
```

RETURNING COMPLEX VALUES

A Python function can return any type of value — most obviously, an integer, float, string, list, tuple, or dict.

What if you want to return more than one value? For example, what if a function takes a string as input, and wants to return the number of vowels versus nonvowels? One option is to return a dict:

```
def count_vowels_nonvowels(text):
    counts = {'vowels':0, 'nonvowels':0}

    for one_character in text.lower():
        if one_character in 'aeiou':
            counts['vowels'] += 1
        else:
            counts['nonvowels'] += 1

    return counts
```

The above function works just fine:

```
count_vowels_nonvowels('hello')    # returns {'vowels': 2,
'nonvowels': 3}
count_vowels_nonvowels('goodbye?') # returns {'vowels': 3,
'nonvowels': 5}
```

What if you want the function to return not only the dict containing the vowel-nonvowel count, but also the original text itself? That means returning two distinct values — and while Python is flexible, a function can only return one value at a time.

Fortunately, one of Python's core data structures is a tuple, which is designed to contain values of different types. You cannot return two values, but you can return one tuple containing two elements. Here is the preceding program, rewritten to return a two-element tuple — the original (input) text, and the counting dict:

```
def count_vowels_nonvowels(text):
    counts = {'vowels':0, 'nonvowels':0}
```

```
    for one_character in text.lower():
        if one_character in 'aeiou':
            counts['vowels'] += 1
        else:
            counts['nonvowels'] += 1

    return text, counts
```

Notice that because `return` is not a function, you don't need to put parentheses around its arguments. Moreover, a comma between two Python values automatically creates a tuple, regardless of whether you have put `()` around them.

```
count_vowels_nonvowels('hello')    # ('hello', {'vowels': 2,
'nonvowels': 3})
count_vowels_nonvowels('goodbye?') # ('goodbye?',
                                     {'vowels': 3,
'nonvowels': 5})
```

Thanks to tuple unpacking, the caller can then retrieve these return values into separate variables:

```
word, vowel_counts = count_vowels_nonvowels('hello')
print(f'{word=}, {vowel_counts=}')
word='hello', vowel_counts={'vowels': 2, 'nonvowels': 3}

word, vowel_counts = count_vowels_nonvowels('goodbye?')
print(f'{word=}, {vowel_counts=}')

word='goodbye?', vowel_counts={'vowels': 3, 'nonvowels': 5}
```

Returning a tuple is a standard way to get around the "only return one value" limitation. Combined with tuple unpacking, it effectively allows you to return any number, and any combination, of values from your functions.

Exercise: Counting Characters

In this exercise, you'll write a function, `count_chars`, that counts the number of characters in a file. By default, it'll count the number of vowels — but by passing a second, optional argument, you'll be able to count other characters instead.

Specifications

Here is what you need to do for this exercise:

- Write a function, `count_chars`, which takes two arguments:
  - The first, `filename`, is a string, the name of a text file.
  - The second, `to_count`, is also a string, the characters that should be counted. By default, this will be the vowels, `'aeiou'`.
- The function should iterate over the file, one line at a time.
- Within each line, it should then iterate over each character, one at a time.
- The function returns a two-element tuple:
  - The first element of the returned tuple is an integer, the total count of matches found.
  - The second element of the returned tuple is a dict, whose keys are the characters in `to_count`, and whose values are integers, the number of times each character appears in the file.

Here are two examples of invoking the function, and then the output:

```
count_chars('passwd.txt')
(2106, {'a': 551, 'e': 701, 'i': 380, 'o': 268, 'u': 206})

count_chars('passwd.txt', 'abcde')
(1840, {'a': 551, 'b': 219, 'c': 154, 'd': 215, 'e': 701})
```

You can solve this exercise interactively, with the help of an AI tutor.

AI Challenge: Strategy Session

This function includes a number of topics that you've already learned about in this book — but sometimes, combining them all into a single function can be particularly challenging. It's not always obvious where to start.

For this exercise, I want you to do a strategy session with GenAI, before you even start to write any code. Talk about how you want to solve this problem — the parameters, the default arguments, the output values, and the loops. See what an AI chatbot has to say. Then, after you've satisfied yourself and the chatbot, go ahead and write the code.

AI Challenge: Prompt and Solution

Here's the start of such a conversation I had with ChatGPT:

```
I'm going through an intro Python book. In a chapter on
functions, I
see the exercise:

In this exercise, you'll write a function, `count_chars`, that
counts
the number of characters in a file. By default, it'll count the
number
of vowels--but by passing a second, optional argument, you'll be
able to count other characters instead.

* Write a function, `count_chars`, which takes two arguments:
** The first, `filename`, is a string, the name of a text file.
** The second, `to_count`, is also a string, the characters that
   should be counted. By default, this will be the vowels,
`'aeiou'`.
* The function should iterate over the file, one line at a time.
* Within each line, it should then iterate over each character,
one at
  a time.
* The function returns a 2-element tuple:
** The first element of the returned tuple is an integer, the
total
   count of matches found.
```

** The second element of the returned tuple is a dict, whose keys are
   the characters in `to_count`, and whose values are integers, the
   number of times each character appears in the file.

Here are two examples of invoking the function, and then the output:

```
count_chars('/etc/passwd')
(2184, {'a': 562, 'e': 726, 'i': 401, 'o': 285, 'u': 210})

count_chars('/etc/passwd', 'abcde')
(1893, {'a': 562, 'b': 222, 'c': 159, 'd': 224, 'e': 726})
```

I don't want you to write code for me. But I do want you to evaluate my strategy for writing this function.

For starters, the function will need to take two arguments. But the
second parameter, `to_count`, will need to have a default of `'aeiou'`. This means the function's first line will need to look like:

```
def count_chars(filename, to_count):
```

Following that:

- I'll assign `total=0`
- I'll iterate over `to_count` in a `for` loop, adding a new key-value pair
  to the dict.
- Then I'll have my output variables ready. Now I open the file, using
  `with` and `open`.
- I'll use a `for` loop on the file object to get each line, one at a time.
- I'll use a (nested) `for` loop on the line, to iterate over each
  character.
- I can then say `if one_character in output`, meaning that only if the
  current character is a key in our dict, we'll add 1 to it. We'll also
  add 1 to `total` in such a case.
- Finally, I'll return both `total` and `counts`, separated by a comma.

What do you think?

ChatGPT responded:

Your strategy is *very* solid — it matches the exercise's intent (iterate line-by-line, then character-by-character), and it'll produce
the right outputs.

That said, there are a few important corrections / improvements to
make:

---

1) Default argument / function signature

You wrote:

```python
def count_chars(filename, to_count):
```

That **does not** give `to_count` a default. If you want the default
vowels, it needs to be:

* `to_count='aeiou'`

So conceptually: yes, 2 parameters — but the 2nd is optional via a
default.

---

2) Initializing the dict from `to_count`

This is a great idea, and exactly what I'd recommend.

Two small notes:

* Make sure keys start at 0 for *each* character in `to_count`.

* If `to_count` contains duplicates (unlikely, but possible), initializing via a loop will just overwrite the same key again

—
  which is fine, but it's worth being aware of. (Not a real
concern in
  an intro exercise.)

---

3) Your membership test is good — but use the right container

You wrote:

> `if one_character in output`

That works perfectly, because `counts` is a dict and membership
checks
test keys.

That said, **this line is a subtle "Python skill test"**:

* `if ch in output` is good (dict membership is fast).
* `if ch in to_count` is *also correct* but will be slower
(string
  membership checks are linear).

So your idea to check membership in the dict is actually *better*
than
checking in the string. Nice.

---

4) Be careful about what counts as "vowels"

Your approach will count only lowercase vowels.

So:

* `'A'` won't count
* `'E'` won't count

That's probably fine given the exercise, but if you wanted to be
robust you'd normalize with `.lower()` before checking. (Not
required
unless specified.)

---

5) File iteration / loops

Everything you propose here is exactly aligned with the instructions:

* `with open(...) as f:`
* loop over lines
* loop over chars in each line

So ✅

---

6) Return value

You wrote:

> return both `total` and `counts`, separated by a comma

Perfect: that produces the required 2-element tuple.

---

7) One more subtle thing: newline characters

When you iterate over characters in a line, you'll also see:

* `'\n'` at the end (usually)

But since you only increment when the character is in the dict keys,
it won't matter unless the user includes `'\n'` in `to_count`.

So that's fine.

Bottom line

✅ Your algorithm is correct
✅ Your data structures are appropriate
✅ Your membership check is efficient
⚠️ Only fix: you must actually give `to_count` a default value in the function
signature

If you want, I can also suggest a few *extra-credit* refinements (still without writing the full solution), such as how to handle upper/lower case, or how to validate input.

Now for the solution. For starters, the `count_chars` function needs two parameters. The first of these parameters, `filename`, is mandatory, but the second, `to_count`, is optional, with a string (`'aeiou'`) as the default value. The first line of the function's definition will thus be:

```
def count_chars(filename, to_count='aeiou'):
```

Next, the function will return two values, an integer and a dict, in a tuple. This means setting up `total`, an integer, at the top of the function, setting it to 0. But it also means setting `counts`, a dict, at the start of the function. Its keys should be the characters in `to_count`, and its values will be 0. A `for` loop over the characters in `to_count` can do that:

```
def count_chars(filename, to_count='aeiou'):
    total = 0
    counts = {}

    for one_character in to_count:
        counts[one_character] = 0
```

With the output variables in place, the function next needs to open the file for reading using `with`, to ensure that the file is closed in an orderly fashion. It'll then iterate over every line in the file with a `for` loop, getting each line as a string. Within that `for` loop, it'll iterate over every character in `each_line`.

That's where things finally get exciting:

- If `one_character` is a key in `counts` (or a value in `to_count`), then `total` will be incremented by 1.
- In addition, `counts[one_character]` — the value in `counts` associated with `one_character` — will be incremented by 1.

```
def count_chars(filename, to_count='aeiou'):
    total = 0
    counts = {}
```

```
    for one_character in to_count:
        counts[one_character] = 0

    with open(filename) as f:
        for one_line in f:
            for one_character in one_line:
                if one_character in counts:
                    counts[one_character] += 1
                    total += 1
```

The final line of the function is a `return` statement. It returns both `total` and `counts` as a tuple — but as per Python convention, there aren't any parentheses around these values:

```
def count_chars(filename, to_count='aeiou'):
    total = 0
    counts = {}

    for one_character in to_count:
        counts[one_character] = 0

    with open(filename) as f:
        for one_line in f:
            for one_character in one_line:
                if one_character in counts:
                    counts[one_character] += 1
                    total += 1

    return total, counts
```

Warning: Don't Use Mutable Defaults!

There's one final, subtle point that's important to know about default argument values: they should be immutable.

As you've seen, Python checks that a function has been invoked with the right number of arguments. If the function has three parameters, then it needs to be called with three arguments. However, that's not the case if there is a default argument value. In that case, Python grabs the value from the defined function, and assigns it to the parameter.

On its face, this doesn't sound like it would be a problem: Python retrieves the default value, assigns it to the parameter, and the function runs.

But if the default value is *mutable*, such as a list or dict, and if the function changes that mutable value, then the data isn't just changed for that particular invocation of the function. The default value is changed for future invocations, too!

This is because default values are set when the function is defined, not when it's invoked. So having an empty list, `[]`, as a default argument value doesn't mean that the parameter will get an empty list each time. Rather, it means the parameter will get *that* list each time. And if it is modified during one invocation? That same list will be around for the next invocation, and will reflect the change.

```
def add_one(a_list=[]):
    a_list.append(1)
    return a_list
```

If you invoke the preceding code with an existing list, it does what you might expect:

```
mylist = [10, 20, 30]

add_one(mylist)
print(mylist)  # [10, 20, 30, 1]

add_one(mylist)
print(mylist)  # [10, 20, 30, 1, 1]
```

If you call `add_one` twice in a row with no argument, though, things get weird and bad:

```
add_one()  # returns [1]
add_one()  # returns [1, 1]
add_one()  # returns [1, 1, 1]
```

That's right — each time you invoke `add_one` without any arguments, the same list — the one created when you defined the function — is assigned to

`a_list`. You end up appending to the same list repeatedly. This leads to terrible confusion and hard-to-debug problems.

For this reason, most Python code checkers and editors will notice mutable defaults, and will tell you that they're a bad idea.

Bottom line: use immutable defaults, and this problem disappears entirely. For example, you could use Python's special `None` value, which cannot be confused with anything else:

```
def add_one(a_list=None):

    if a_list == None:
        a_list = []

    a_list.append(1)
    return a_list
```

Because `[]` is created each time the function runs, rather than when it is created, it cannot be carried over across invocations.

*args

You might have discovered that `print` can not only display any Python value, but that it can take multiple arguments. Not a list of values, but it can actually take any number of arguments:

```
print('a')              # prints a
print('a', 'b')         # prints a b
print('a', 'b', 'c')    # prints a b c
```

You might say that this is possible with a very large number of parameters, each with a default value. But `print` doesn't have any maximum number of arguments, and no one would seriously suggest defining a function with several thousand parameters, each with a default of `''` or the like.

Instead, the solution is to define a function such that if it is invoked with more arguments than there are parameters, the unassigned positional

arguments should all be put into a tuple. The function can then iterate over that tuple, using each of its values.

Traditionally, that tuple is called `args`, although there isn't any rule that you call it that. And to tell Python that this is a special parameter, one that should contain a tuple with all otherwise unassigned arguments, you put a `*` before it when defining the function. Programmers commonly call the `*` character "splat," and thus the `*args` syntax is often known as "splat args." (Wondering why it's called "splat"? Because `*` looks something like a bug after you have stepped on it.)

For example, consider this function:

```
def myfunc(x, y, *args):
    return f'{x=}, {y=}, {args=}'
```

Notice that `*args` is the final parameter named in the function. It will always be a tuple, and it will always contain the unassigned arguments, in the order that they were passed.

> **NOTE**
>
> Actually, `*args` won't always be the final parameter in a function. You will soon learn about *keyword arguments*, and Python supports a number of special parameter types having to do with them — all of which come after `*args`.

What happens if you call `myfunc`? It depends on how many arguments you provide:

- If you invoke `myfunc(10)`, then you'll get an error. That's because you must provide values for both `x` and `y`.
- If you invoke `myfunc(10, 20)`, then `10` is assigned to `x`, and `20` is assigned to `y`. What about `args`? All of the arguments were assigned to parameters, so `args` remains empty, `()`.

- If you invoke `myfunc(10, 20, 30)`, then `10` is assigned to `x`, and `20` is assigned to `y`. `args` is a tuple containing the single value `30`, or `(30,)`. Remember that a one-element tuple will always have a trailing comma inside the `()`.
- If you invoke `myfunc(10, 20, 30, 40)`, then `10` is assigned to `x`, and `20` is assigned to `y`. `args` is a two-element tuple, `(30, 40)`, with the arguments that weren't assigned to any other parameter.

For example, consider a function that takes a list of integers, sums them, and returns the result:

```
def mysum(numbers):
    total = 0

    for one_number in numbers:
        total += one_number

    return total
```

There isn't anything wrong with this `mysum` function: it iterates over `numbers`, one element at a time, adds each to `total`, and then returns `total`.

To invoke the function in this way, you would have to write:

```
mysum([10, 20, 30, 40, 50])
```

With `*args`, you can just pass the numbers as individual, separate arguments:

```
def mysum(*numbers):    # *numbers, rather than *args
    total = 0

    for one_number in numbers:
        total += one_number

    return total
```

To invoke this version of the function, you can then say:

```
mysum(10, 20, 30, 40, 50)   # [] are removed!
```

The main reasons to use `*args`, and not take a list of arguments, are aesthetics and convenience. It can feel more natural to invoke a function with a bunch of arguments, rather than pass a list. Just remember to document your function appropriately, since the caller will need to know whether they're passing a single list of values or a number of separate arguments.

Exercise: Counting Characters (Multiple Files)

In the previous exercise, you wrote `count_chars`, a function that took a filename and `to_count`, a string containing the characters you wanted to count. The function returned two values, the total count and a dictionary indicating how many times each character in `to_count` appeared.

You'll now modify the function such that the caller can pass more than one filename. The function should total the counts for all of the files.

Specifications

Reimplement `count_chars`, with the following differences:

- The first parameter is now `to_count`, the string containing characters to be counted.
- No longer will `to_count` have a default value; the user will need to specify which characters to count.
- All subsequent arguments will now be treated as filenames to be read.

- The total character count, as well as the counts in the dictionary, will represent the total found across all files.
- Print the name of each file as it is processed, for debugging purposes.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The first change that will need to happen to `count_chars` is in its first line, where the parameters are declared. The previous version had two parameters, `filename` and `to_count`. In this version, `to_count` doesn't have a default argument value. In addition, `*args` needs to be the final parameter, to catch all of the otherwise unassigned arguments. The first line of the function's new version will thus be:

```
def count_chars(to_count, *args):
```

Not much else needs to change in the program, until it gets to opening and iterating over the file: before, when `filename` was a string containing a filename, you could just invoke `open` to get the file object, and then iterate over each line in the file. But in this version, `args` is a tuple of filenames. The program thus needs to iterate over each filename, opening it in sequence.

This means, believe it or not, that you need three nested `for` loops:

- The outer `for` loop goes through each file.
- The middle `for` loop goes through each line in the current file.
- The inner `for` loop goes through each character in the current line.

Here's how that can look:

```
def count_chars(to_count, *args):
    total = 0
```

```
    counts = {}

    for one_character in to_count:
        counts[one_character] = 0

    for one_filename in args:                    # loop over
filenames
        print(f'Processing {one_filename}...')
        with open(one_filename) as f:
            for one_line in f:                   # loop over lines
                for one_character in one_line:   # loop over chars
                    if one_character in counts:
                        counts[one_character] += 1
                        total += 1

    return total, counts
```

Because of the new, outermost `for` loop, the `with` statement and everything inside its block need to be indented one additional level inward. Other than that, the function doesn't change. And the `return` statement, which returns `total, counts`, doesn't need to change at all.

Note that inside the outermost `for` loop, the function calls `print`. It's normally a bad idea to have `print` inside a function, but when the function will run for a long time, it can be useful for updates, especially when developing and debugging the code.

AI Challenge: Reverse Socratic Method

The Reverse Socratic Method technique is great for investigating and understanding new topics. In this case, you should use this technique to pepper the GenAI system with questions about this program — how it works, what techniques it uses, and even how else it could have been implemented.

Here's an example interaction, with Gemini:

```
I'm reading an introductory Python book, and am currently
learning
about functions. In the latest exercise, I wrote a function,
`count_chars`:
```

```
def count_chars(to_count, *args):
    total = 0
    counts = {}

    for one_character in to_count:
        counts[one_character] = 0

    for one_filename in args:                   # loop over
filenames
        print(f'Processing {one_filename}...')
        with open(one_filename) as f:
            for one_line in f:                  # loop over lines
                for one_character in one_line:  # loop over chars
                    if one_character in counts:
                        counts[one_character] += 1
                        total += 1

    return total, counts
```

I'm going to ask you a lot of questions about this code, so that I can
understand it better.

My first question: These three nested loops seem excessive. Yes, we
have to go through each file, but must we really go through each line
in each file, and then each character in each line?

Gemini responded:

It's a great question. When you see three levels of indentation, your
"this feels complex" alarm should definitely go off — that's a sign of a
good programmer's intuition!

However, in this specific case, the nesting is actually **necessary
and logical** because of how data is structured within a computer's
memory and file system.

Why the Nesting Exists

To understand why we need all three, we have to look at the "hierarchy" of the data you are trying to process:

1. **Level 1 (`args`):** You have a collection of files. You can't count characters in "File B" until you've opened it, so you must iterate through the list of filenames.

2. **Level 2 (`f`):** Python (and most operating systems) reads files most efficiently line-by-line. This prevents your computer from running out of memory if you try to open a massive 10GB file all at once.

3. **Level 3 (`one_line`):** Since your goal is to count **individual characters**, you eventually have to "touch" every single character to see if it matches what you're looking for.

Can We "Flatten" It?

Technically, yes, you can make the code *look* shorter, but the computer still has to do the same amount of work.

For example, you could use the `.read()` method to skip the "line" loop:

```python
for one_filename in args:
    with open(one_filename) as f:
        content = f.read()  # This reads the ENTIRE file into memory at once
        for one_character in content:
            if one_character in counts:
                # ... count logic ...
```

The Trade-off: While this removes one loop from your screen, it's actually "riskier." If `one_filename` is a tiny text file, it's fine. If it's a massive log file, your program might crash because

```
  it's trying to cram millions of characters into your RAM at
  once. Your original "line-by-line" approach is the **standard
  professional practice** because it is memory-efficient.

### The Reality of Performance

In Big O notation (the way we measure algorithm speed), this is ,
where is the total number of characters across all files. Whether
you
nested them or not, you cannot count characters without looking
at
each of them at least once.

> **Note:** The first loop in your code (the one initializing the
    `counts` dictionary) is separate from the others. That one is
    totally fine — you're just setting the stage!

Would you like me to show you how to use a `Counter` from
Python's
standard library to make this whole process much more concise?
```

Warning: Defaults and *args

One last word of warning about `*args`: it's a bad idea to have any parameters with default argument values in a function that also has `*args`. Remember that `args` gets the arguments that weren't assigned to any other parameter. But because arguments are assigned in order, and because `*args` must be the final parameter in your function's signature, putting values in `args` means passing through — and thus assigning to — the parameters with defaults.

For example:

```python
def myfunc(x, y=10, *args):
    return f'{x=}, {y=}, {args=}'
```

Now, invoke the function with different numbers of arguments:

```python
print(myfunc(2))            # x=2, y=10, args=() -- y is default,
args is empty
print(myfunc(2, 3))         # x=2, y=3, args=() -- y is 3, args
is empty
```

```
print(myfunc(2, 3, 4, 5))    # x=2, y=3, args=(4,5) -- y is 3,
args is (4,5)
```

In order to assign values to `args`, you must also assign a value to `y`. There isn't any way to populate `args` with arguments and leave `y` with its default value.

For this reason, it's a bad idea to have parameters with defaults in a function that also accepts `*args`. There are ways to avoid this problem, such as *keyword-only parameters*, but those are a bit complex and beyond the scope of this book.

Keyword Arguments

You've already seen that when you call a function with arguments, those values are assigned to parameters in order: the first argument is assigned to the first parameter, the second to the second parameter, the third to the third parameter, and so forth.

For example:

```
def greet(first_name, last_name, city):
    return f'Hello, {first_name} {last_name} from {city}!'
```

This `greet` function has three parameters, none of which have default argument values. You must invoke the function with three arguments:

```
greet('Reuven', 'Lerner', 'Modiin')
```

As you would expect, the arguments are then assigned to parameters, in order:

- `'Reuven'` is assigned to `first_name`
- `'Lerner'` is assigned to `last_name`
- `'Modiin'` is assigned to `city`

Not surprisingly, these are known as *positional arguments*.

Python supports a second type of argument, known as a *keyword argument*, in which the caller specifies the parameter to which the argument will be assigned. If the preceding call to `greet` were written using keyword arguments, it would look like this:

```
greet(first_name='Reuven', last_name='Lerner', city='Modiin')
```

Passing Keyword Arguments

As you can see, every keyword argument consists of a parameter name, an = sign, and the value that should be assigned to that parameter. When you call a function with keyword arguments, the position no longer matters. For that reason, you could also call `greet` in this way:

```
greet(city='Modiin', last_name='Lerner', first_name='Reuven')
```

Keyword arguments solve a number of problems. First, they tell Python very explicitly which argument should be assigned to which parameter. That makes the code that calls the function clearer and less ambiguous. This is especially true if a function has many parameters; using keyword arguments reduces the chance that you'll get lost when writing or reading the invoking code.

Second, if a function has one or more parameters with default argument values, keyword arguments make it possible to override some of those defaults, but leave others in place. For example, consider a function that produces a report based on some data. The function definition could begin with:

```
def format_report(data, title='Report', columns='all',
                  sort_by=None, ascending=True, max_rows=50,
                  include_totals=False, date_format='%Y-%m-%d')
```

You could invoke this function in a number of ways, using keyword arguments to override the defaults, while keeping the rest in place:

```
format_report(data, title='Annual report', max_rows=10,
include_totals=False)
format_report(data, title='Monthly report', sort_by='date',
              max_rows=12, date_format='%m %d, %Y')
format_report(data, sort_by='department', ascending=False,
include_totals=False)
```

As you can see, a combination of defaults and keyword arguments gives you a great deal of flexibility in both defining and invoking a function.

Keyword Argument Gotchas

You can mix positional and keyword arguments when calling a function. However, all of the positional arguments must come before all of the keyword arguments. For example, consider this `greet` function:

```
def greet(first_name, last_name, city):
    return f'Hello, {first_name} {last_name} from {city}!'
```

You can invoke it as:

```
greet('Reuven', 'Lerner', city='Modiin')
```

This code works, because both positional arguments come before the keyword argument. Trying to pass any keyword arguments before any positional arguments will result in a Python syntax error:

```
greet(first_name='Reuven', 'Lerner', city='Modiin')
```

Python's error message even spells out the problem:

```
SyntaxError: positional argument follows keyword argument
```

NOTE

It's easy to confuse the default argument values with keyword arguments, since both involve functions, and both involve the use of `name=value` syntax. Remember that default argument values are set in the function *definition*, in the list of parameters that comes after `def`. Keyword arguments, by contrast, are passed when you *invoke* the function, assigning values to parameters explicitly, rather than letting those assignments take place positionally.

Exercise: Fancy Calculator

In this exercise, you will take the `calc` function that you wrote earlier, and change it such that two parameters (`operator`, and `second`) have default argument values. You'll then invoke the function a number of times, each passing a different combination of positional and keyword arguments.

Specifications

Here is what you are to do:

- Modify `calc` such that two of the parameters, `operator`, and `second`, get default argument values. You can decide what those values should be.
- Now invoke `calc` a number of times, with different combinations of positional and keyword arguments. Be sure to print the result of each invocation on the screen, so that you can see how the invocation affects the result.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

You needed to change the `calc` function definition such that `operator` and `second` had default argument values. This implementation makes `'+'` (addition) the default operator, and `0` (an integer) the default value for `second`. The rest of the function remains unchanged:

```
def calc(first, operator='+', second=0):
    if operator == '+':
        result = first + second
    elif operator == '-':
        result = first - second
    else:
        result = '(unknown)'
    return f'{first} {operator} {second} = {result}'
```

The addition of two default argument values means that you can now invoke `calc` in a number of different ways:

```
calc(10, '+', 20)                   # returns '10 + 20 = 30'
calc(10, second=20)                 # returns '10 + 20 = 30'
calc(10, operator='-')              # returns '10 - 0 = 10'
calc(10, second=5, operator='+')    # returns '10 + 5 = 15'
calc(10)                            # returns '10 + 0 = 10'
calc(first=10)                      # returns '10 + 0 = 10'
```

AI Challenge: Find the Bug

For this AI challenge, give a GenAI system both the new `calc` function and all of the example invocations with positional and keyword arguments. Ask the AI system to insert a bug into the code, either in the function or in the lines calling it. Can you find the bug?

Here is an example transcript from working with Claude:

```
I'm reading an intro Python book, and am learning about
functions. I wrote
the following function:

def calc(first, operator='+', second=0):
    if operator == '+':
        result = first + second
    elif operator == '-':
        result = first - second
    else:
        result = '(unknown)'
    return f'{first} {operator} {second} = {result}'

I can then invoke the function in any of the following ways:
```

```
calc(10, '+', 20)
calc(10, second=20)
calc(10, operator='-')
calc(10, second=5, operator='+')
calc(10)
calc(first=10)

I want you to challenge me, and ensure that I understand this
material, by
inserting a bug into the code -- either the function definition
or one
of the invocations. Let's see if I can find the bug you insert!
```

Claude responded:

```
Great idea, Reuven — this is a classic way to solidify
understanding. Here's your code with one bug introduced.
Everything
else is untouched:

```python
def calc(first, second=0, operator='+'):
    if operator == '+':
        result = first + second
    elif operator == '-':
        result = first - second
    else:
        result = '(unknown)'
    return f'{first} {operator} {second} = {result}'

calc(10, '+', 20)
calc(10, second=20)
calc(10, operator='-')
calc(10, second=5, operator='+')
calc(10)
calc(first=10)
```

Your mission: **Which invocation(s) will now produce incorrect or
unexpected counts, and why?**

Take your time — try tracing through the calls mentally before
running
anything.
```

I wrote:

```
Aha, you switched the order of the parameters. Any invocation of
calc with
keyword arguments will continue to work just fine, as will
relying on the
default argument values. But positional arguments will now be
interpreted
incorrectly.
```

Local Versus Global Variables

You've already seen that assigning to a variable replaces its previous value:

```
x = 1234
print(f'First, x = {x}')

x = 5678
print(f'Then, x = {x}')

x = 9123
print(f'Finally, x = {x}')
```

But what happens if you invoke two functions, both of which assign to the variable `x`?

```
def first():
    x = 1234

def second():
    x = 5678

x = 9999

print(f'Before, x = {x}')
first()
second()
print(f'After, x = {x}')
```

You might be surprised to discover that both before and after calling `first` and `second`, the variable `x` has a value of `9999`. The assignments inside `first` and `second` had no effect on the variable `x`.

This is actually considered a good thing! If assigning to a variable in a function were to affect the variable's value in other functions, it would be harder to understand and debug our programs. Or you would need to ensure that every function in a program used unique variable names — a particularly difficult thing to do when many programmers work on a project.

For these reasons, Python divides variables into two categories:

- *Global* variables are defined outside a function. A global variable is created when you assign to it outside a function body.
- *Local* variables are defined inside a function. A local variable is created when you assign to it *inside* a function body.

Note that in the following example, `x` is a global variable, but `y` is a local variable, inside my `myfunc` function:

```
x = 1234          # outside a function, so x is global

def myfunc():
    y = 5678      # inside a function, so y is local
```

Trying to use a local variable outside a function will result in an error.

You can see a graphical depiction of local versus global variables in Figure 10-1.

But what happens if you try to use a global variable inside a function? It'll work just fine:

```
x = 1234          # outside a function, so x is global

def myfunc():
    y = 5678      # inside a function, so y is local
    print(f'In myfunc, x = {x} and y = {y}')
```

Invoking `myfunc()` displays:

```
In myfunc, x = 1234 and y = 5678
```

You can see a graphical depiction of how a function can access both local and global variables in Figure 10-2.

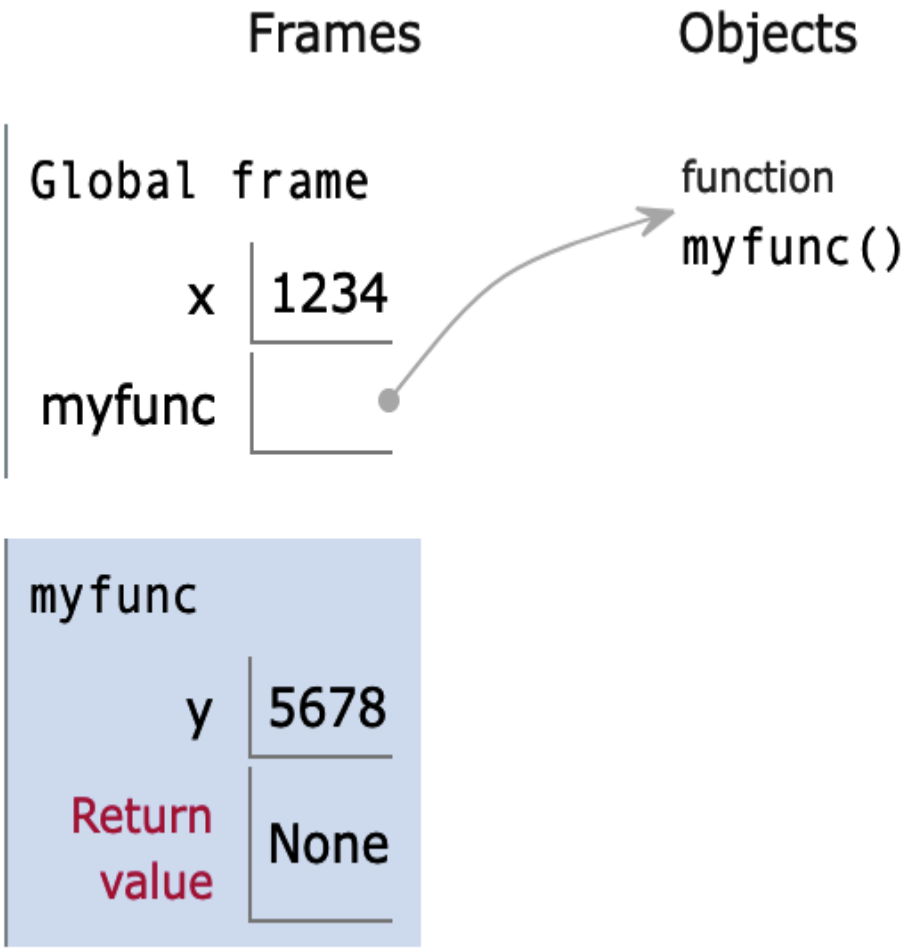


Figure 10-1. In this screenshot from the Python Tutor, you can see that `x` is a global variable defined in the global frame, while `y` is a local variable defined within the `myfunc` frame

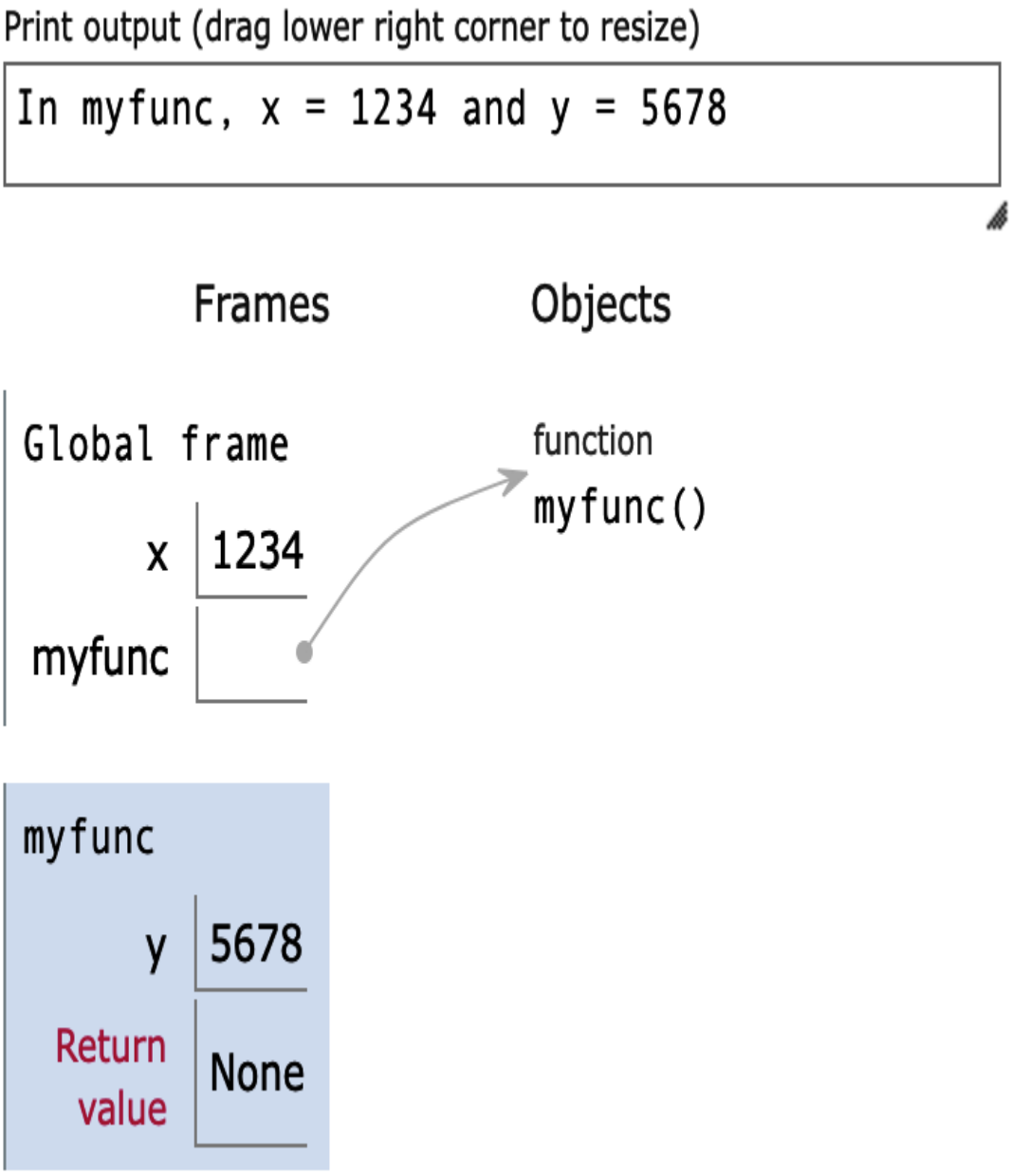


Figure 10-2. In this screenshot from the Python Tutor, the `myfunc` function is able to print both `x` (global, defined outside the function) and `y` (local, defined inside the function)

The trouble starts when you're in a function, and there is a local variable with the same name as a global one. In such a case, Python will always give priority to the local variable:

```
x = 1234          # this x is global

def myfunc():
```

```
    x = 5678    # this x is local

    print(f'In myfunc, x = {x}')
```

If you run `myfunc()`, you will get the output

```
In myfunc, x = 5678
```

In this case, the local `x` variable is said to *shadow* the global `x` variable. That can be a problem if you want to use the global `x` inside the function.

You can see a depiction of how a local variable "shadows" (i.e., hides) a global varaible of the same name in Figure 10-3.

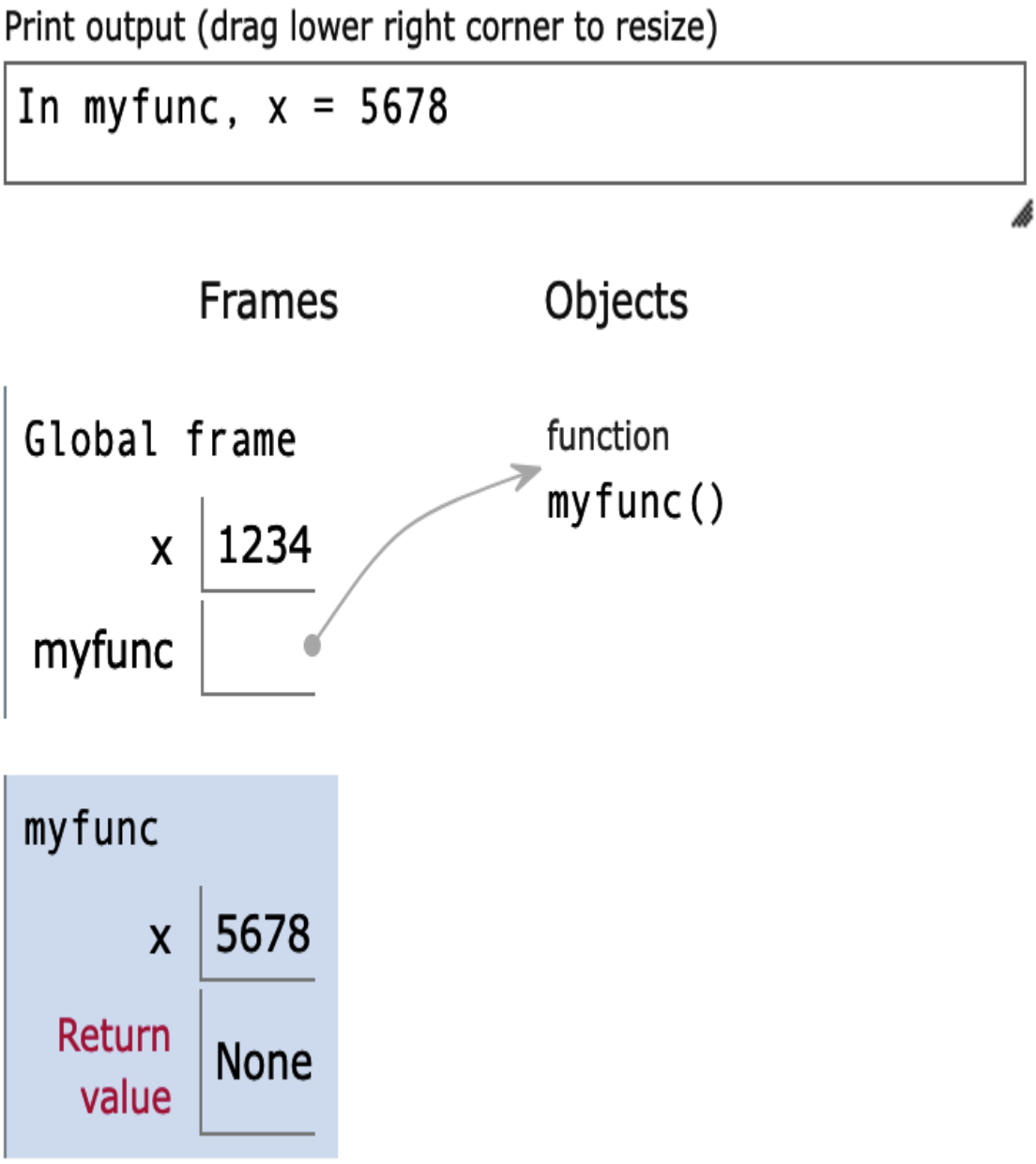


Figure 10-3. In this screenshot from the Python Tutor, the `myfunc` function tries to print `x`. It only sees the local `x`, which shadows the global one.

As a general rule, try to avoid using the same variable names in your functions as exist outside of functions. Python won't complain, but it can give you all sorts of surprising results. And anyway, using long, descriptive variable names reduces the chance that you'll have such conflicts.

Summary

In this chapter, you deepened your understanding of functions: you learned how to document them with docstrings, so that other developers will know what the function does, what it expects, and what it returns. You saw how to give parameters default argument values. And you saw how keyword arguments can make function calls less ambiguous and more flexible.

Given how much of a developer's time is spent writing functions, it's no surprise that you have gone through two long chapters on how to write and use them. But where do you store these functions, especially when you want to reuse them in other programs? In the next chapter, you'll learn about modules and packages, allowing you to apply the "don't repeat yourself" ("DRY") rule to larger chunks of code.

Chapter 11. Modules and Namespaces

You've already heard about DRY, namely "Don't repeat yourself," twice in this book:

- First, when you first learned about loops in Chapter 5. Instead of repeating the same code on several consecutive lines, you can use a loop.
- Second, when you learned about functions in Chapter 9. Instead of repeating the same code in several places in a program, you can write the code in only one place, and then refer to it by invoking the function multiple times.

The DRY principle helps to keep your code shorter, cleaner, easier to understand, and easier to maintain. It's a good thing to keep in mind not only when you're writing code, but also when you're fixing and maintaining it.

There is a third situation in which DRY applies: what if the same functionality is needed and used in multiple programs? For example, you might have several programs that need to perform the same calculation, or implement the same login-security strategy, or read from a particular type of file. It would make sense to write such code a single time, and then use it in whichever program needs that functionality.

This isn't a new idea, and most programming languages support it using something known as libraries. A *library* allows you to define a function or data structure, and put it in a shared location. Whenever a program needs that particular functionality, it can use the existing code from the library. Using an existing library, rather than writing the code yourself, has a number of advantages:

- Once a problem has been solved, you don't need to spend time and effort solving it again, reinventing the wheel.
- Your program becomes shorter, easier to write, easier to understand, and easier to maintain.
- If someone else has written a library, then you don't have to write or maintain that code. (However, you might need to maintain the code that uses their library, which can also be challenging.)
- A shared library will typically be faster and have fewer bugs than any new code you write, because it has evolved and improved over time.
- You gain the advantage of abstraction, only need to think about the name and functionality, rather than the implementation.

There is another advantage to libraries, namely that it's easier for people to work on smaller tasks. If you had to conceptualize a large software project all at once, it might well be overwhelming. And even if you could do it, remembering each piece and what it does, it would be hard to focus on only one piece at a time. By breaking a project into separate pieces, each implemented in its own library, it becomes easier to think about and plan, as well as to divide the work among a number of people or teams.

It's even common to create your own library, if you expect to use the same code in a number of different programs.

In the Python world, we call our libraries *modules*. Actually, the terminology is a little more complex than that:

- A Python *module* is an individual file that defines data structures and functions.
- A Python *package* is a collection of modules.

If you're just using modules, rather than implementing them, then the difference between modules and packages isn't that important. To a large degree, you can use the terms interchangeably.

Python's modules also provide *namespaces*, which you can think of as surnames for variables. In Chapter 10, you learned how variables inside a function are *local*, meaning that they only exist within the context of that function. This means that multiple functions can use the same variables without having to worry that another function is using the same variable name. Similarly, if one module defines a variable `x` and another module also defines a variable `x`, the two `x` variables will be kept separate by virtue of the fact that they're defined in separate modules — much as two children with the same first name can have different last names.

Python comes with a large collection of modules, referred to (somewhat confusingly) as the *standard library*. You'll use a handful of those modules in this chapter, as you learn how to use `import`. In Chapter 12, you'll learn more about Python's standard library — what it contains, and how you can use it. And in Chapter 13, you'll learn about the vast number of third-party packages available on PyPI, the Python Package Index, and how you can download, install, evaluate, and use those packages.

By the end of this chapter, you'll understand how to use Python's modules. While some aspects of Python programming can be complex or hard to understand, modules tend to be one of the more straightforward topics, complicated mostly by the enormous number of modules from which you can choose.

Using a Module

To tell Python that you want to use a module in your program, use the `import` statement. For example, to use the `random` module, which provides functionality having to do with random numbers, you would say:

```
import random
```

Once you have done that, `random` is defined as a global variable. However, it isn't a string, list, or dict. Rather, it is a *module* value. Such a

value doesn't really do much itself; you can think of it as a container, or even a warehouse, for storing other values and functions.

There are a few surprising things to notice about `import`:

- `import` isn't a function. It's a statement, like `for` or `with`. For that reason, you don't use `()` after its name, as you would with a function or method.
- The name after `import` isn't a filename, and thus isn't a string — and doesn't have quotes around its name. It is a variable, which will refer to the module value after it is loaded.
- For this reason, you cannot use `import` to point Python to a particular filename. (You'll see, in Chapter 12, where Python looks for modules you import.)

What's in a Module?

The data structures and functions defined in a module are available as *attributes*, names that come after a period (`.`). If a module defines a function, you can invoke it by naming the module, `.`, and then the function you want to run.

> **NOTE**
>
> You've already used methods (e.g., `str.strip` and `list.append`), which are one specific type of attribute. Attributes are Python's way of storing one value inside another, whether it's a function in a module or a method in an object type. *Every* value in Python has attributes, and each attribute can refer to anything — a string, dict, function, or even a module. Modules are special, in that they *only* contain other values, and aren't really trying to do anything else.

For example, the `random` module has a function, `randint`, that returns a random integer within a range. This code generates and prints a random integer between 1 and 100:

```
import random

n = random.randint(0, 100)
print(n)
```

Notice that you cannot just invoke `randint` on its own; the name only exists within the `random` module's namespace, as `random.randint`. Other than that, it's a regular function, one that takes arguments and returns a value.

A similar method, `random.choice`, returns one randomly selected element from a sequence (i.e., string, list, or tuple). For example:

```
letter = random.choice('abcd')
print(f'Random letter is {letter}')

number = random.choice([10, 15, 20, 25])
print(f'Random number is {number}')
```

Generating random numbers, and choosing random elements from a sequence, might seem like simple tasks. But they are actually quite complex to get mathematically correct. The functions in Python's `random` module, because they have been developed, checked, and used by millions of developers over several decades, are less likely to contain bugs than anything you (or any individual) might write.

Exercise: Guess the Name

In this game, the user needs to correctly choose a randomly selected name from a list.

Specifications

Here's what you have to do for this exercise:

- Define a list of names (strings), `names`. (For example, you might define it to be the list of planets, or of people in your family.)

- Use `random.choice` to select one name at random. Assign it to `secret`.
- Define an empty list, `already_guessed`.
- Define `guess_counter` to be the integer 1. This will count how many guesses the user needed to find the secret value.
- With those in place, repeatedly ask the user to guess a name.
    - If they guess the secret name, stop asking and report how many guesses it took.
    - If they guess a name that is *not* in `names`, then scold them and let them try again.
    - If they guess a name that they have already tried (and is thus in `already_guessed`), then scold them, and let them try again.
    - If they guess a name that is in `names`, but isn't `secret`, then add that name to `already_guessed` and increment `guess_counter`.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The program needs to start with `import random`. Without that line, the module won't be loaded, and the `random` variable won't be defined. Trying to access anything in the `random` module will thus result in a `NameError` exception.

Once `random` has been loaded, the program defines a few variables. In this version, `names` is set to a list of eight strings, the names of planets in the solar system. `secret` is then set to a randomly chosen value from `names`, thanks to `random.choice`:

```
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1
```

There isn't any way to know how many times the user will make a guess. As a result, it's best to use a `while True` loop here. Inside the loop, the program then uses `input` to ask the user to guess which name (planet) has been chosen:

```
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1

while True:
    guess = input('Enter guess: ').strip()
```

There are a few different scenarios here, each handled with an `if` statement. First, perhaps the user will guess the secret name. In that case, the program congratulates them, and tells them how many guesses it took. (That's why the `guess_counter` variable needed to start with a value of 1.) The `break` within this `if` block ensures that the program exits as soon as the user guesses correctly.

```
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1
```

```
while True:
    guess = input('Enter guess: ').strip()
    if guess == secret:
        print(f'You guessed {secret} in {guess_counter}
guesses!')
        break
```

But what if the user doesn't guess correctly? Then there are a few other options:

First, perhaps the user has already guessed this name. In such a case, we won't hold it against them, but merely give them a gentle scolding and reminder.

How can we know if the user has already guessed this name? That's what the `already_guessed` list is for, containing all of the names that could have been `secret`, but weren't, and that the user has guessed already. The program uses `in` to search in the `already_guessed` list; if `in` returns `True`, then the program scolds them, and then has them guess again. Notice that a repeated incorrect guess doesn't increment `guess_counter`. Also note the use of `continue` to go back to the top of the loop body:

```
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1

while True:
    guess = input('Enter guess: ').strip()
    if guess == secret:
        print(f'You guessed {secret} in {guess_counter}
guesses!')
        break
    if guess in already_guessed:
        print(f'You guessed {guess} already; try again')
        continue
```

Another possibility is that the user has guessed a name that isn't in `names`. In such a case, the program should scold the user, but in a different way from before — no longer indicating that this is a repeated (incorrect) guess, but instead that this guess isn't legal. This time, the `in` operator isn't being run on `already_guessed`, but instead on `names`:

```
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1

while True:
    guess = input('Enter guess: ').strip()
    if guess == secret:
        print(f'You guessed {secret} in {guess_counter}
guesses!')
        break
    if guess in already_guessed:
        print(f'You guessed {guess} already; try again')
        continue
    if guess not in names:
        print(f'{guess} is not a legal guess; try again')
        continue
```

The final possibility is that `guess` *is* in `names`, but is not the secret, and hasn't been guessed before. In such a case, the user gets a mild scolding, and (crucially) `guess` is added to the `already_guessed` list. In addition, `guess_counter` is incremented by 1:

```
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1

while True:
```

```
    guess = input('Enter guess: ').strip()
    if guess == secret:
        print(f'You guessed {secret} in {guess_counter}
guesses!')
        break
    if guess in already_guessed:
        print(f'You guessed {guess} already; try again')
        continue
    if guess not in names:
        print(f'{guess} is not a legal guess; try again')
        continue

    print(f'Sorry, but {guess} is not the secret name.')
    already_guessed.append(guess)
    guess_counter += 1
```

The program exits the `while` loop when the user has correctly guessed `secret`, using an f-string to print both `secret` and the number of guesses that the user needed:

```
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1

while True:
    guess = input('Enter guess: ').strip()
    if guess == secret:
        print(f'You guessed {secret} in {guess_counter}
guesses!')
        break

    if guess in already_guessed:
        print(f'You guessed {guess} already; try again')
        continue

    if guess not in names:
        print(f'{guess} is not a legal guess; try again')
        continue

    print(f'Sorry, but {guess} is not the secret name.')
```

```
        already_guessed.append(guess)
        guess_counter += 1

print(f'You guessed {secret}. Guesses needed: {guess_counter}')
```

NOTE

Is it better to use `if`/`elif`/`else`, or three `if` statements in a row? There isn't any one answer, but there is a significant semantic difference: when you use `if`/`elif`/`else`, you're explicitly saying that one, and only one, of those can be true. With several `if` statements in a row, none of them might fire. Or, by contrast, more than one might — or even all of them. Given that this name-guessing program should only allow for one option, why did the solution here use `if` several times in a row? Because each `if` block ended with either `break` or `continue`. The only way to get to the next `if` (let alone the `print` at the end of the program) would be for the previous conditions to be `False`.

AI Challenge: Add a Bug

Once you have finished solving this exercise, you should modify the code, planting a bug somewhere in it. Predict what the problem is with this code. Will a GenAI system find and explain the problem correctly? For that matter, can you predict what will go wrong when you add this bug?

Here is what I gave to Gemini:

```
I'm reading an intro Python book. For the chapter on modules, I
wrote the
following program:

--
import random

names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn',
         'Uranus', 'Neptune']
secret = random.choice(names)
already_guessed = []
guess_counter = 1

while True:
    guess = input('Enter guess: ').strip()
    if guess == secret:
```

```
        print(f'You guessed {secret} in {guess_counter}
guesses!')
        continue

    if guess in already_guessed:
        print(f'You guessed {guess} already; try again')
        continue

    if guess not in names:
        print(f'{guess} is not a legal guess; try again')
        continue

    print(f'Sorry, but {guess} is not the secret name.')
    already_guessed.append(guess)
    guess_counter += 1

print(f'You guessed {secret}. Guesses needed: {guess_counter}')
--

Actually, this isn't the program I wrote. I added a bug to it.
Can you
find that bug, and explain what will go wrong?
```

Gemini responded:

```
It looks like you've set a clever trap for yourself! You've
created an
**infinite loop** that prevents the game from ever actually
finishing.

### The Bug: The `continue` keyword

The issue lies within the first `if` block of your `while True`
loop:

```python
if guess == secret:
    print(f'You guessed {secret} in {guess_counter} guesses!')
    continue  # <--- This is the culprit

```

What goes wrong?

When the user correctly guesses the secret planet:
```

```
1. The program prints the success message.

2. It hits the `continue` statement.

3. `continue` tells Python to skip the rest of the current loop
and
jump straight back to the **start** of the `while` loop.

4. The user is prompted to "Enter guess" again, even though they
already won.

Because there is no `break` statement to exit the loop, the final
`print`
statement at the bottom of your script will **never be
executed**, and the game
will never end unless the user forces the program to quit (like
hitting Ctrl+C).

---

### The Fix

To fix this, you need to replace `continue` with `break` when the
guess is
correct. This tells Python to "break out" of the loop entirely
and move on
to the code following it.

```python
    if guess == secret:
        print(f'You guessed {secret} in {guess_counter}
guesses!')
        break  # This exits the while loop
```
```

Alternative Forms of Import

Python comes with a `math` module, which defines a number of mathematical functions and variables, including `math.pi`, a float with the (approximate) value of pi. If you're writing a program that uses pi, you could just refer to `math.pi` repeatedly. At a certain point, that might feel a bit long-winded. Maybe you could just use `pi` instead? Python will understand what you mean, right?

Actually, it won't:

```
import math

print(pi)  # error
```

The result will be a `NameError` exception, indicating that Python doesn't know about the name `pi`. It knows about the name `math`, a module with an attribute `pi`, so that you can refer to `math.pi`. But `pi` all by itself? Python hasn't ever heard of it.

This particular example might seem silly, because both `math` and `pi` are relatively short names. But the point remains, that when you `import` a module, you are only defining one variable, the module itself. Any names that it contains must be accessed via that module, as an attribute.

While this makes logical sense, it's also a bit annoying. Surely there must be a way to access `pi` directly, without mentioning `math` each time?

And indeed, Python provides an alternative form of `import`, `from .. import`, for situations like this one:

```
from math import pi

print(pi)  # now it works!
```

Here's how to use `from .. import`:

- The statement has two parts. In the first (after `from`), you mention the module (`math`, in this case) that should be loaded. In the second (after `import`), you indicate the variable from the module that should be defined.
- If it hasn't been imported before, then the module is loaded into memory.
- The named attribute from the module is then defined as a global variable.

- The module itself is *not* defined as a variable.

You can import more than one name from a module with `from .. import`, separating the names to import with commas:

```
from math import pi, isqrt
```

The preceding code ensures that both `pi` and `isqrt` are defined as global variables.

> **NOTE**
>
> Many Python developers believe that `from .. import` saves memory, because only the name that they explicitly requested will be loaded into memory. This isn't true. The entire module is loaded into memory, but only the named variable is defined and accessible. Use `from .. import` so that you can refer to variables more directly and succinctly, not because you believe it'll save memory.

The biggest downside of using `from .. import` is that it only gives access to the names you specifically imported. If you want short-form access to some names, but still have (long-form) access to the rest of the names in a module, use `import` on the module and `from .. import` on a subset of names in that module:

```
import math            # define global variable math
from math import pi    # define global variable pi
```

The preceding code imports `math`, giving access to all of its values and functions. But it also defines `pi` as a global variable, allowing you to write `pi` instead of `math.pi`.

While `from .. import` is convenient, there is at least one possible downside: without the explicit module name, you might forget where it was defined, or whether it was defined in a module at all. You will have to decide whether it's better to have longer code that retains the variable's context and origin, or whether shorter code is better. There are, however,

software-development teams that have decided not to permit the use of `from .. import` for this reason.

Aliasing When Importing

Some modules have long names. Some functions defined inside modules have long names. And sometimes, a module name might conflict with the name of a variable or function that you have already defined.

In all of these cases, you can use another form of `import`, namely `import .. as`. For example:

```
import math as m
```

In this case, the `math` module will be imported, but `math` will not be defined as a variable. Rather, the `m` variable will be defined. Again, this can be useful if a module (or its contents) has a long name.

There is another common use case: certain modules, especially in the data-science world, have a tradition of being imported with certain aliases. For example:

```
import numpy as np     # NumPy always gets the "np" alias
import pandas as pd    # pandas always gets the "pd" alias
```

There isn't a rule that you have to create these aliases, but it's so standard at this point that it might as well be required.

Just as you can alias a module when using `import`, you can also alias a specific name that you have imported from a module when using `from .. import`. For example:

```
from math import pi as p
```

The preceding code will:

- Load the `math` module, if it hasn't been loaded yet.

- Assign the global variable `p` to refer to the `pi` name inside the `math` module.
- It will *not* define either `math` or `pi` as global variables, however.

To summarize, there are four ways that you can import a module into Python:

- `import` *`module`* imports the module, and defines *`module`* as a global variable
- `import` *`module`* `as` *`alias`* imports the module, but defines *`alias`* as a global variable referring to it
- `from` *`module`* `import` *`name`* imports the module, but only defines *`name`* as a global
- `from` *`module`* `import` *`name`* `as` *`alias`* imports the module, and defines *`alias`* as a global variable referring to *`name`* in *`module`*

How Not to Import

There is a fifth way to import from a module, namely `from MODULE import *` — but really, you should avoid using it unless you have absolutely no choice. This version of `import` tells Python that it should import the module into memory, but that *every* name defined in the module should be defined as a global variable.

There are a number of reasons why importing all of the names is a terrible idea. First and foremost, programmers have fought against global variables and in favor of namespaces for decades, because namespaces make it easier to organize and understand a program. Putting all of the names into a single basket makes things more chaotic. It also means that if and when a module changes, adding or removing names that it has defined, your program's global variables will change, too. This might mean surprise collisions

between the variables you define and those defined in the module you import.

Imagine, for example, the chaos that would ensue if everyone were to suddenly no longer have surnames. It would be far more complex to distinguish people from one another, especially people with popular names. In the same way, using `from .. import *` tells Python, "I'm sure that this module doesn't define any variables that will conflict with ones I've already written." That's a risky thing to do, especially when loading a module you didn't write and don't maintain.

For all of these reasons, avoid using `from .. import *` except under extreme circumstances. For example, some modules expect to be loaded in this way, and will break if they aren't. Even then, see what you can do to import particular names, rather than asking for all of them at once.

Exercise: Password Generator

In this exercise, you will generate a new random password using four pools of characters — lowercase letters, uppercase letters, digits, and punctuation. The user will decide how many random characters will be chosen from each pool. The program will then put those randomly selected characters together into a new password.

Specifications

- From the `string` module, load several names:
  - `ascii_lowercase` (aliased to `lc`)
  - `ascii_uppercase` (aliased to `uc`)
  - `digits` (no alias)
  - `punctuation` (aliased to `punc`)
- From `random`, load:

  - `shuffle` (aliased to `shuf`)
  - `choices` (no alias)
- Define `pw_characters`, an empty list.
- Ask the user, in order:
  - How many lowercase letters do they want?
  - How many uppercase letters do they want?
  - How many digits do they want?
  - How many punctuation characters do they want?
- Using `random.choices`, retrieve the appropriate number of characters from each source string, and add to `pw_characters`.
- Invoke `random.shuffle` on `pw_characters` to mix them up.
- Use `str.join` to turn the list into a string.
- Print the resulting password.

You can work through a version of this exercise with an AI-based Socratic tutor.

AI Challenge: Strategy Session

For this exercise, the GenAI task is a strategy session, meaning that you should run your ideas by the chatbot before you start to code. As a result, the GenAI task is presented here *before* the exercise solution. Think about what you want to do, and how you want to do it, and ask GenAI what it thinks about those steps. As usual with a strategy session, it's important to tell the chatbot *not* to write code or solve the problem for you.

AI Challenge: Prompt and Solution

Here is an example session with Gemini:

```
I'm going through an intro Python book, and I'm currently on the
chapter about modules. Here is an exercise.
```

Here, I inserted the exercise description from before.

```
I want to run my planned strategy by you, to see what you think
about
my plan. I don't want you to write any code, but rather tell me
where
my planned solution could be improved (or that it's good as is).

Here's my plan:

- First, I'll use "from string import ascii_lowercase as lc",
also
  getting and aliasing uc and punctuation, but not aliasing
digits.
- Then I'll use "from random import" to get both shuf and
choices.
- Then I'll call "input" four times, each time asking for a
different
  numeric input.
- Then I'll invoke "choices" (which I got from "random") four
times,
  each with a different input string (from the "string" module)
and a
  different number (from what the user gave me). I'll be sure to
  invoke "int" on the user's input, to convert it first.
- As I get each list back from "choices", I'll add it to the
output
  list--which started as an empty list, but grew over time.
- I'll then use += to add the return value from choices to
output,
  giving me a longer and longer list of characters.
- I'll use "shuf" (from random) to mix the characters up, so that
  they're truly randomized, including mixing around the different
  groups
- Finally, since that will give me a list of one-character
strings,
  I'll use ''.join() to get a string back, the final password.

What do you think?
```

Gemini replied:

```
Your strategy is solid and follows the exercise requirements
precisely. You
have a clear grasp of how the `string` and `random` modules
interact, and
your plan for data type conversion (using `int`) and list
management is spot on.

Here are a few observations and minor refinements to consider as
you
move from the plan to the code:

### 1. The Import Syntax

Your plan for the `string` module imports is perfect. For the
`random`
module, just ensure your syntax follows the same pattern: `from
random
import shuffle as shuf, choices` This keeps your namespace clean
and
matches the aliasing requirements of the exercise.

### 2. Handling List Growth

You mentioned starting with an empty list and using `+=` to add
the
results of `choices`. This is a very "Pythonic" way to do it.
Since
`choices` returns a list, and `+=` on a list expects an iterable,
it
will effectively "extend" your `pw_characters` list by appending
each
individual character rather than nesting lists.

### 3. The `shuffle` Gotcha

One detail to remember when you write the code: `random.shuffle`
(or
your alias `shuf`) **mutates the list in place** and returns
`None`.

* **Correct:** `shuf(pw_characters)` then use the list.
* **Incorrect:** `password = shuf(pw_characters)` (This will
  result in your password being `None`).

### 4. Input Validation (Optional but Good Practice)
```

```
While the exercise doesn't explicitly demand it, keep in mind
that
`input()` returns a string. If a user accidentally types "five"
instead of "5", your `int()` conversion will throw a
`ValueError`. If
you want to make the script robust, you could wrap those inputs
in a
simple try/except block, though for a basic exercise, your
current
plan is perfectly sufficient.

### 5. String Joining

Using `''.join(pw_characters)` is the standard, most efficient
way to
turn a list of characters into a single string. You've correctly
identified that `shuf` leaves you with a list, and `join` is the
bridge back to a string format.
---

**Overall Verdict:** Your logic is sound and you've accounted for
the
  necessary type conversions and method behaviors. You're ready
to code it!
```

Let's now address the solution. To start off, this program requires the use of two different modules, `string` and `random`. And while you *could* just use these modules as they are, some of the names they use are a bit long. So for the purposes of this exercise, you needed to use `from .. import`, giving aliases to some (but not all) of the names needed from those modules.

If you chose to have several separate invocations of `from .. import`, once for each imported name, that would be fine. But you can also import multiple names on a single line. Each can optionally have an alias:

```
from string import ascii_lowercase as lc, ascii_uppercase as uc
from string import digits, punctuation as punc
from random import shuffle as shuf, choices
```

Next, the program created `pw_characters`, and asked the user how many of each password should be chosen:

```
pw_characters = []
lc_number = input('How many lowercase letters? ').strip()
uc_number = input('How many uppercase letters? ').strip()
d_number = input('How many digits? ').strip()
s_number = input('How many symbols? ').strip()
```

It's from this point that things get a bit trickier: the `random.choices` function, which here was imported as just `choices`, takes one positional argument, a sequence. It also takes one keyword argument, `k`, with an integer indicating how many random characters should be retrieved from that sequence. So saying `choices('abcd', k=3)` will return a list of three randomly chosen characters from `'abcd'`. (And yes, they might repeat.)

Also remember that because `input` returns a string, the user's input will need to be turned into an integer with `int`:

```
pw_characters += choices(lc, k=int(lc_number))
pw_characters += choices(uc, k=int(uc_number))
pw_characters += choices(digits, k=int(d_number))
pw_characters += choices(punc, k=int(s_number))
```

Notice that the code here used += to add elements of a list (the return value from `choices`) to an existing list (`pw_characters`). After these four lines run, `pw_characters` is a list of randomly selected password characters.

Well, they're *fairly* random: but the lowercase letters come before the uppercase letters, which come before digits, which come before symbols. For this reason, you needed to then invoke `random.shuffle`, imported here as `shuf`, to mix them around:

```
shuf(pw_characters)
pw = ''.join(pw_characters)
print(pw)
```

Here is the full program:

```
from string import ascii_lowercase as lc, ascii_uppercase as uc
from string import digits, punctuation as punc
from random import shuffle as shuf, choices

pw_characters = []
lc_number = input('How many lowercase letters? ').strip()
uc_number = input('How many uppercase letters? ').strip()
d_number = input('How many digits? ').strip()
s_number = input('How many symbols? ').strip()

pw_characters += choices(lc, k=int(lc_number))
pw_characters += choices(uc, k=int(uc_number))
pw_characters += choices(digits, k=int(d_number))
pw_characters += choices(punc, k=int(s_number))

shuf(pw_characters)
pw = ''.join(pw_characters)
print(pw)
```

You can solve this exercise interactively, with the help of an AI tutor.

Summary

Used correctly, modules reduce the amount of code you need to write and maintain, while increasing program speed and reducing bugs. In this chapter, you learned to use `import` in a variety of ways, using modules such as `random`, `string`, and `math`. All three of these modules are part of Python's *standard library*, which comes with Python and ensures you have access to common, useful functionality. In the next chapter, you'll explore and learn about the standard library, and understand how to use it in your own code.

Chapter 12. Python's Standard Library

When you install Python, you aren't just downloading the language. You're also installing a large collection of modules that are considered important enough that they come with Python, and are installed alongside it. This collection is known as the *standard library*, and with rare exceptions works on every platform. If you use a module from the standard library in your Python program, you know that it'll already be installed and available wherever the program runs.

You already used the `math`, `random`, and `string` modules from it in Chapter 11, and saw how useful they can be. In this chapter, you'll dive deeper into the standard library. You'll learn where Python looks for modules, and how the standard library fits into that search path. You'll also look at the contents of a module, and at two useful modules in the standard library, `collections` and `os`. And you'll learn how to use the standard library's documentation to improve your understanding and coding.

The Standard Library

New major versions of Python are released every October. This is documented via special Python Enhancement Proposal documents, known as PEPs. Each release includes a new version of the Python language itself, typically with new features, bug fixes, and speedups. It also includes the standard library. Indeed, while most people primarily pay attention to core language changes, the standard library is where most of the changes really take place.

That's partly because the standard library is so large, and partly because it's used in so many Python programs. Even a small speedup in one standard library module might affect millions of computers.

The standard library is large not only because it solves so many types of problems, but because it provides a standard baseline for all Python installations. You don't need to worry about whether someone's Python installation includes support for dates, or URL retrievals, or reading CSV documents; every Python installation includes all of this, and more. For years, the Python community liked to say that the language is "batteries included," because the standard library removes the need to download and install third-party solutions for so many things.

That's no longer as true as it used to be; a large number of popular packages are distributed independent of the standard library. And over the last few years, Python's core developers have been working to remove no-longer-needed modules from the standard library, described as "dead batteries." But because the standard library is so widely used, the removal of any module from the standard library is a long, careful process that takes place over many years and many Python releases.

If your Python program imports one or more modules from the standard library, then you know it'll work on any other computer running the same version of Python, regardless of platform. So if you're developing software on a Mac, and it uses modules from the standard library, then all of your users, including those on Windows and Linux, will be able to use it.

Why, if the standard library is included with Python, do you still need to `import` its modules? Shouldn't they just be loaded automatically? No, because loading a module takes time and consumes memory. If Python were to load the entire standard library, then it would take quite a long time for a program to start up. Plus, the loaded data structures and functions would all consume a great deal of memory, even if they weren't ever used.

Python thus expects you to become somewhat familiar with the standard library, choosing and loading its contents as needed. That's easier said than done; the standard library is huge, and just remembering what it contains can be a daunting task. Fortunately, the documentation is online. It's worth looking at the online documentation, both to learn what the standard library

contains, and also to understand how to use its various modules and functions.

Where Does import Look for Modules?

When you say `import random`, the name `random` isn't a string or a filename. Rather, it's the name of the variable you want to assign. In that sense, you can almost think of `import name` as `name = import(name_module)`. But that just begs the question: what is `name_module`? What is being loaded? Is there a file somewhere? If so, where?

A module, at the end of the day, is a file containing Python code. Python program files traditionally use a *.py* suffix — and as you might have suspected, `import name` tells Python to look for a file called *name.py*. So the name you hand to import plays double duty, both indicating what variable should be defined and also what file should be loaded. (So the name you give to `import` isn't a filename, but is used to construct a filename.)

But where is this file located? Python actually looks in a number of places:

- First, it looks in the current directory, where "current" describes the program that is being run. If you're using Jupyter, then it'll look in the directory in which you started Jupyter.
- If it doesn't find the file there, it looks through all of the module directories that came with Python, the *standard library*. This typically consists of three directories:
    - A ZIP file containing all of the files in the standard library. You can think of this file as a directory in disguise.
    - A directory containing all of these files, unzipped.
    - A *lib-dynload* directory, containing binary modules written in compiled languages such as C, C++, and Rust

— typically for cases when Python code would be too slow, and using a faster-to-execute, harder-to-code-in language will be useful.

- If Python doesn't find the file in the standard library, it looks through *site-packages*, a special directory into which third-party modules from PyPI are installed. (You'll learn more about PyPI and `pip` in Chapter 13.)
- If Python cannot find a file for your module in any of these places, it gives you a `ModuleNotFoundError`.

How does Python know to look in each of these places? With a list of strings, known as `sys.path` — that is, the `path` variable defined in the `sys` module that comes with Python.

> **NOTE**
>
> The `sys` module describes many aspects of the running Python interpreter, including the version (`sys.version`), the platform (`sys.platform`), and command-line arguments (`sys.argv`).

The `sys.path` list is set when Python starts up. You can see the precise directories with a bit of Python code:

```
import sys
for one_dir in sys.path:
    print(f'{one_dir=}')
```

Here is the output from one computer, with long lines broken apart:

```
one_dir=''
one_dir='/Users/reuven/.local/share/uv/
    python/cpython-3.14.0-macos-aarch64-none/lib/python314.zip'
one_dir='/Users/reuven/.local/share/uv/
    python/cpython-3.14.0-macos-aarch64-none/lib/python3.14'
one_dir='/Users/reuven/.local/share/uv/
    python/cpython-3.14.0-macos-aarch64-none/lib/python3.14/lib-
dynload'
```

```
one_dir='/Users/reuven/.local/share/uv/
    python/cpython-3.14.0-macos-aarch64-none/lib/python3.14/site-
packages'
```

The first element of `sys.path` is the empty string, meaning that Python should search in the current directory. (That's the blank line in the preceding output.) Then come three directories for the standard library. Then, finally, comes the *site-packages* directory.

You can ask a module in which file it was defined, by examining its `__file__` attribute. This is a string. For example:

```
import random
print(random.__file__)
```

This code returned the (very long) string `'/Users/reuven/.local/share/uv/python/cpython-3.14.0-macos-aarch64-none/lib/python3.14/random.py'`. The filename (at the end) is *random.py*, matching the name of the module, `random`. That module was in one of the directories in `sys.path`, as you might expect, given that `random` is in the standard library.

Guess what, though? You don't need to worry about this very much. Almost all of the modules you use will either come with Python (in the standard library) or will be a third-party module installed via `pip` or `uv`, and thus in `site-packages`. (You were introduced to `pip` back in Chapter 1. Chapter 13 goes into more detail, and introduces the modern `uv` tool that many people are using in place of `pip`.) For the most part, this all works like magic, and it's fine to treat it as such. Use `import`, name the module you want to load, and don't worry about where it is located or how module names translate to filenames.

What's in a Module?

So far, you've seen how to `import` a module into Python, and then use one of the names provided by the imported module. But a module typically defines a number of variables and functions. How can you know what those are, and what they do?

One way to answer this question is with the built-in `dir` function, which returns a list of strings, the names of attributes available in the module. For example, you can see the names available in `random` with:

```
import random

print(dir(random))
```

This returns a (very long) list of strings:

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_ONE', '_Sequence',
 '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__',
'_accumulate',
 '_acos', '_bisect', '_ceil', '_cos', '_e', '_exp', '_fabs',
'_floor',
 '_index', '_inst', '_isfinite', '_lgamma', '_log', '_log2',
'_os',
 '_parse_args', '_pi', '_random', '_repeat', '_sha512', '_sin',
 '_sqrt', '_test', '_test_generator', '_urandom', 'betavariate',
 'binomialvariate', 'choice', 'choices', 'expovariate',
'gammavariate',
 'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'main',
 'normalvariate', 'paretovariate', 'randbytes', 'randint',
'random',
 'randrange', 'sample', 'seed', 'setstate', 'shuffle',
'triangular',
 'uniform', 'vonmisesvariate', 'weibullvariate']
```

You might recognize some of the names in this list. The string `'randint'` means that you can invoke `random.randint(0, 100)`. And the string `'choice'` means that you can run `random.choice('abc')`.

This list does have a number of limitations: first and foremost, you cannot know which names refer to data and which refer to functions. Moreover,

beyond taking educated guesses regarding the names, you can't know what each function does. This technique is best if you are already somewhat familiar with a module, and just need to jog your memory.

You saw in Chapter 10 that functions can have docstrings; it turns out that modules can have them, too. In a notebook or other interactive environment, you can view the module's docstring by invoking `help`:

```
help(random.choice)
```

Here's the output from this function:

```
choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
```

The length of a docstring can vary considerably. This is a pretty short one, but many of the docstrings in the pandas data-analysis package are long and thorough.

The best documentation, especially for modules in the standard library, is at *https://oreil.ly/H6BAy*. It's an invaluable resource — not just because it contains complete documentation on every data structure, function, and class defined in standard library modules, but also because it often comes with example code and helpful advice. For example, the documentation for `random.choice` is:

```
random.choice(seq)

    Return a random element from the non-empty sequence seq. If
seq
    is empty, raises IndexError.
```

This is slightly longer and more specific than the docstring. But beyond that short example, the documentation includes a "recipes" section, where `random.choice` is used in several example programs.

You might have noticed that the list of attributes in `random` includes a number of names that include underscore characters (_), and others that are

in ALL CAPS. These reflect some naming conventions in the Python world that you should learn to recognize:

- An attribute name in ALL CAPS, such as `random.BPF`, is referred to as a *constant*, meaning something that should be treated as permanent. (Notice that while the module name is all lowercase, the attribute is in all caps.) This is a convention, so while you should not assign to constants, the Python language sees them as variables, and assigning to them will work.
- A name that starts with one underscore, such as `random._parse_args`, is considered *private*, meaning that while you can use it, it is really meant for the module's own internal use. Again, this is a convention; you *can* use such a name, but the module's author is signaling that the name might change at any time, and that doing so is risky.
- A name that both starts and ends with two underscores, for example, `random.__file__`, is known among Python coders as a *dunder*, with *dunder* being shorthand for "double underscore." These names are special; some of them (such as `__file__`) are set by Python and are how the language leaves reminders for itself when executing a program. Others (such as `__all__`) are defined in a module, and affect Python's behavior. You will, over time, learn how to use some of these dunder names, and also when to define your own. As a general rule, avoid defining any dunder names unless you know what you're doing. If you accidentally define a name that affects Python's behavior, it'll be very hard to debug.

If you stick with naming your variables and functions with lowercase letters, putting _ characters between words, then you won't conflict with these names, and won't accidentally change Python's behavior.

Exercise: Exploring a Module

In this exercise, you will explore the `math` module in Python's standard library. Along the way, you'll not only learn about its functionality, but also how to learn about a new module.

Specifications

This exercise involves very little coding, but a fair amount of exploring and playing:

1. Import the `math` module, from Python's standard library.
2. How many names are defined? How many names that do *not* start or end with an underscore (`'_'`) does it define?
3. Ask the user to enter the radius of a circle. Print the area of that circle, given that the area can be calculated as the square of the radius times `math.pi`.
4. Ask the user to enter a large number. Using the module's documentation, find the function that calculates the integer square root of a number. (That is, the square root without any fractional part.) Calculate the integer square root of the user's input.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

To answer these questions, you needed to import the `math` module from Python's standard library. That meant saying:

```
import math
```

Notice, once again, that importing a module is done with `import` followed by a variable name — the name of the module you want to import.

How many names does the `math` module define? Those are all attributes, and you can get a list of attributes with the built-in `dir` function. And as

you might remember, the `len` function returns the number of elements in a list. You can thus print the number of attributes in the `math` module with:

```
print(len(dir(math)))
67
```

Many of those names start or end with `'_'`, the underscore character. How can you count the number of `math` attributes that *don't* start or end with `'_'`? One way is to define a variable, `non_underscore_counter`, to be 0, then to run a `for` loop over the attributes, checking that the first and final characters aren't underscores:

```
non_underscore_counter = 0
for one_name in dir(math):
    if one_name[0] != '_' and one_name[-1] != '_':
        non_underscore_counter += 1
print(non_underscore_counter)
62
```

The preceding code iterates over each name (string) in `dir(math)`, one at a time. It then checks that neither `one_name[0]` (the first character in the string) nor `one_name[-1]` (the final character in the string) is an underscore, `'_'`. If that's the case, then `non_underscore_counter` is incremented by 1. The final number of non-underscore names in `math` is 62.

Next, you were to ask the user to enter a radius, and then (using the pi-r-squared formula you might remember from middle school) calculate the area of a circle with that radius. Here's one option:

```
radius = input('Enter radius: ')
Enter radius: 5

area = math.pi * int(radius) ** 2
print(area)
78.53981633974483
```

Notice that the preceding code doesn't check that the user's input only contained digits, which is fine for something small like this. The user entered a radius, and that radius was used in a calculation. Since `math.pi` is defined as a float, multiplying it by an integer will result in a float.

Finally, you needed to ask the user to enter a large number, and then to calculate the integer square root of that number. The instructions didn't tell you what function to use, in order to encourage you to check on the Python documentation site. The function you needed was `math.isqrt`, and here is one way to use it:

```
large_number = input('Enter a large number: ').strip()
Enter a large number: 123456789
print(math.isqrt(int(large_number)))
11111

print(11111 ** 2)
123454321

print(11112 ** 2)
123476544
```

Notice that the answer it provided, `11111`, is not quite the square root of `123456789`. But it is the largest integer whose square is not greater than `123456789`. Squaring the next integer, `11112`, returns `123476544`, which would be too high.

Here is the complete code, along with its output:

```
import math

print(len(dir(math)))
67

non_underscore_counter = 0
for one_name in dir(math):
    if one_name[0] != '_' and one_name[-1] != '_':
        non_underscore_counter += 1
print(non_underscore_counter)
62

radius = input('Enter radius: ')
```

```
Enter radius: 5

area = math.pi * int(radius) ** 2
print(area)
78.53981633974483

large_number = input('Enter a large number: ').strip()
Enter a large number: 123456789
print(math.isqrt(int(large_number)))
11111

print(11111 ** 2)
123454321

print(11112 ** 2)
123476544
```

AI Challenge: Interview

For the GenAI task this time around, enter everything you've done so far and have the chatbot interview you about the `math` module, the functions and constants you used, and how you used them. For example, I wrote to ChatGPT:

```
I'm reading an intro Python book. I'm on the chapter about
modules.
It gave me the following challenge:
```

> (I entered the exercise specifications here, along with my solution code.)

```
In order to improve my understanding, please interview me about
this
code. Ask me a question, then I'll respond, and you should tell
me
what I got right and wrong. Keep doing this until I tell you to
stop.
```

ChatGPT responded:

```
Great — this is an excellent way to deepen your understanding.
I'll act
like an interviewer and we'll go step-by-step. I'll ask **one
question
at a time**, wait for your answer, then give feedback and ask the
next
question.
---

### Question 1: Understanding `dir(math)`

You wrote:

```python
print(len(dir(math)))
```

and got `67`.

Interview question:

What exactly does `dir(math)` return? Be precise. What kind of
object
is it, what does each element represent, and where do those names
come
from?
```

I wrote:

```
math is a module. dir(x) returns a list of strings, the names of
attributes available on whatever x is. So here, dir(math) returns
a
list of strings, the names of attributes available via the math
module. Since that's a list, we can run len on it, getting the
length
of the list.
```

ChatGPT repsonded:

```
You nailed the core idea.

**What you got right**

* ✅ `math` is a module object.
```

* ✅ `dir(x)` returns a **list of strings**.

* ✅ Those strings are the **names you can access as attributes**
    (like `math.pi`, `math.sqrt`, `math.isqrt`, etc.).

* ✅ `len(dir(math))` counts how many such attribute names are present.

A couple of "level-up" details

* `dir()` is *introspection*: it's meant to show what names an object
  makes available. For modules, that's basically the module's
  namespace.

* Where do the names come from?

  * Some are defined/assigned in the module (constants like `pi`,
    functions like `sin`).

  * Many in `math` are implemented in C and exposed into the module
    namespace.

  * Some "dunder" names like `__doc__`, `__name__`, etc., come from
    Python's module machinery.

Small precision note

* `dir()` gives a helpful view of names, but it's not always
  "everything that exists" in a deep sense; it's "what Python chooses
  to report" as the attribute list. For modules like `math`, that's
  pretty close to the full public namespace.

Question 2: Your underscore filter logic

You wrote:

```python
if one_name[0] != '_' and one_name[-1] != '_':
    non_underscore_counter += 1
```

```
**Interview question:**

Your instructions said: "How many names that do *not* start
**or** end
with `_` does it define?"

Does your condition match that English requirement exactly? If
not,
what would be the correct boolean logic — and what difference
would it
make?
```

Examples from the Standard Library

The standard library evolves slowly, but it does change, growing over time to include modules that are considered so useful and important that every Python program should have access. It now contains hundreds of modules, many of which have nothing in common except that they're used by many Python developers and are released along with the Python language.

It would be impractical to tell you about all of the useful modules in the standard library. Instead, you'll now learn about a small subset of what it offers, using those modules to write a Python program that would otherwise have been difficult or time-consuming to write.

collections.Counter

A Python module is an individual file. When several modules are located in the same directory, they're called a *package*. One of the oldest and most popular packages in the standard library is `collections`, which contains a number of useful data structures. One of these is `Counter`, designed to count the number of times values appear in a sequence.

> **NOTE**
>
> Behind the scenes, `Counter` is just a variant form of `dict`. So it supports all of your favorite dictionary functionality, as well as new things that have to do with counting.

For example, you could say:

```
from collections import Counter
counts = Counter('abcbcbcbcdd')
```

The `Counter` object has now counted the number of times that `a`, `b`, `c`, and `d` each appear in that string. These strings serve as the keys, and the number of times each appears is its value. Because `Counter` is a form of `dict`, you can use `dict.items` to get the keys and values:

```
for key, value in counts.items():
    print(f'{key}: {value}')
```

Here is the output:

```
a: 1
b: 4
c: 4
d: 2
```

`Counter` also implements some of its own methods. One of the best is `most_common`, which returns a list of two-element tuples in descending order, from the most common to the least. Thanks to tuple unpacking, you can iterate over it just as you would `dict.items`:

```
for key, value in counts.most_common():
    print(f'{key}: {value}')
```

The output:

```
b: 4
c: 4
d: 2
a: 1
```

You can initialize `Counter` with any iterable, not just a string. For example, here are the forecast high temperatures for Paris over the coming two weeks, put into `Counter`:

```
counts = Counter([13, 14, 13, 12, 13, 12, 15, 18, 19, 18, 19, 19,
19, 19])
```

What three temperatures will occur most often? Invoke `most_common` with an argument:

```
for key, value in counts.most_common(3):
    print(f'{key}: {value}')
```

The results:

```
19: 5
13: 3
12: 2
```

If you have a list of strings or integers, and want to know which occur most frequently, you'll find `Counter` to be particularly useful.

Pathlib

In Chapter 8, you learned how to read from and write to files. `with`, `open`, and the other tools you learned about in that chapter are a great start in working with files. But:

- What if you want to get a list of all files in a directory?
- What if you want to check whether a file is truly a file, or is really a directory?
- What if you want to get information about a file, such as its size?

For years, Python developers used the standard library's `os` package to answer these and other questions. `os` (short for *operating system*) is a standard cross-platform way to work with files, filenames, and directories. In particular, the `os.path` module within `os` provides lots of helpful functionality.

However, a programming language can only do so much to mask the differences between operating systems, especially when it comes to file and directory names. Windows uses `\` (a backslash) to separate directory names and filenames, while Unix (e.g., macOS and Linux) use `/`. The top of a Windows filesystem is typically `C:\`, while on a Unix system, it's `/`. Windows doesn't typically care about the case of filenames, while Unix systems are more sensitive.

Moreover, `os` provided so much functionality that it was often hard to remember everything that it did, even if you were not thinking about different operating systems.

For many Python developers, a modern solution to many of these issues is `pathlib`, a module that is part of Python's standard library. `pathlib` doesn't solve all of these problems, but it does go a long way toward providing a unified, consistent approach that handles many of the most common cases.

The cornerstone of `pathlib` is the `Path` data structure, which you can `import` into your Python program:

```
from pathlib import Path
```

To work with either a directory or a file, you create a new `Path` value, handing it the string — filename or directory name — as an argument:

```
p1 = Path('passwd.txt')
p2 = Path('asdfadfaasdf')
```

Notice that Python is, in both cases, willing to create a `Path` value and assign it to the a variable. However, you can then invoke the `exists` method on that `Path` object, to find out if the file exists:

```
print(p1.exists())  # True
print(p2.exists())  # False
```

Before reading from a file with `open`, you can invoke `exists` to check whether it exists, thus avoiding an error.

What if you want to get a list of all files in a directory? `Path` values can handle directories, too, and support an `iterdir` method, meant to be used in a `for` loop:

```
p3 = Path('.')

for one_filename in p3.iterdir():
    print(one_filename)
```

What if you want to know which of the filenames represent files, and which are directories? Each of the values returned by `iterdir` is itself a `Path`, allowing you to invoke the `is_file` and `is_dir` methods:

```
p3 = Path('.')

for one_filename in p3.iterdir():
    if one_filename.is_file():
        filetype = 'file'
    elif one_filename.is_dir():
        filetype = 'dir'
    else:
        filetype = 'other'

    print(f'[{filetype}] {one_filename}')
```

To get only those filenames that match a particular pattern, you can use the `glob` method. Pass it a string, and it will return all filenames matching that string, with `?` representing any one character and `*` representing one or more characters:

```
p3 = Path('.')

for one_filename in p3.glob('*.txt'):
    print(one_filename)
```

Traditionally, you could get the size of a file by invoking `os.stat` on the file. That would return a special `stat` value with a number of attributes,

including `st_size`, the size of the file. With `pathlib`, you can do the same thing via the `stat` method. Make sure to only ask for the size on a file; requesting it for a directory will result in an error:

```
p3 = Path('.')

for one_filename in p3.glob('*.txt'):
    if one_filename.is_file():
        size = one_filename.stat().st_size
    else:
        size = '(no size available)'

    print(f'{one_filename}, size: {size}')
```

What about opening files? You can invoke `open` on a `Path`, just as you can invoke it on a string (as you learned in Chapter 8). But you can also invoke `open` as a method, even using it within a `with` block:

```
p3 = Path('.')

for one_filename in p3.glob('*.txt'):
    if one_filename.is_file():
        size = one_filename.stat().st_size

        with one_filename.open() as f:
            first_line = f.readline()
    else:
        size = '(no size available)'
        first_line = ''

    print(f'{one_filename}, size: {size}')
    print(f'\t{first_line}')
```

The preceding code invokes the `readline` method, which returns a single line (including the ending newline character `'\n'`) from an open file.

Finally, if you have a `Path` value that refers to a directory, and you want to get a `Path` that refers to a file in that directory, you can combine them using the `/` operator — yes, the same slash that you have already used for division in Python. This works on all operating systems, including Windows, where the traditional separator character is a backslash.

For example, here is an example of how to print (in a slightly roundabout way) the entirety of the *.zshrc* configuration file, stored in my home directory (*/Users/reuven*) on my computer running macOS:

```
p4 = Path('/Users/reuven/')    # Path for my home directory
zshrc = p4 / '.zshrc'          # Use / to get a Path for my .zshrc
config

with zshrc.open() as f:
    for one_line in f:
        print(one_line.rstrip())
```

Notice that the preceding code has two `Path` values:

- The first, `p4`, is the result of calling `Path` on a string containing a directory name.
- The second, `zshrc`, is the result of invoking `/` on `p4` and a string (filename).

Exercise: Using the Standard Library

In this exercise, you will find which words appear most frequently in a directory of text files. You'll be aided by both `pathlib` and by `collections.Counter`.

Specifications

1. Define an empty list, `words`. This list will grow over the course of the program to include all of the words, from all of the files, you read through.
2. Ask the user to enter a directory name.
3. Use `glob` to get the text files (i.e., files ending with `'.txt'`) in this directory. Note that giving a directory name will not be

sufficient; you need to give the full pattern, with both the directory name and `'*.txt'`.

4. Go through each filename, and if it is a regular file, open it. Print the filename, so that the user gets a status update.
5. Go through each line in the file. Break the line into a list of words (strings), and add them to `words`.
6. When the program has gone through all of the text files, create a `Counter` object based on `words`.
7. Use the `most_common` method to display the 10 most common words in these files, and how many times each appeared.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

The program starts by importing `Path` from `pathlib`, and `Counter` from `collections`:

```
from pathlib import Path
```

With those in place, you can then prepare the main variables — `words`, a list that starts empty but which will grow to include all of the words in the file, and `dirname`, which you'll get from the user via `input`:

```
from pathlib import Path
from collections import Counter

words = []
dirname = input('Enter directory name: ').strip()
```

Next, you'll use `Path.glob` to iterate over all of the text files in `dirname`. You can do that by creating a pattern, starting with `dirname` and ending with `'*.txt'`. Iterate over each filename you get back,

invoking `open` on it, printing the name (for easier debugging), and then iterate over the file, one line at a time with a `for` loop.

While iterating over each line of the file, use `str.split` to break each line into a list, and use += to add that list of words to `words`.

Notice that this is a *nested* `for` loop: the outer loop iterates over the filenames you get from `Path.glob`, and the inner loop iterates over lines in the current file, `one_filename`:

```
from pathlib import Path
from collections import Counter

words = []
dirname = input('Enter directory name: ').strip()

for one_filename in Path(dirname).glob('*.txt'):
    with one_filename.open() as f:
        print(one_filename.name)
        for one_line in f:
            words += one_line.split()
```

At this point, `words` contains all of the words from all of the text files in `dirname`. You can create a `Counter` object from this list — a dict-like object in which the words are the keys and the values are the counts of occurrences. You can then invoke `most_common(10)` on the `Counter` object, getting the 10 most common words and their frequencies:

```
from pathlib import Path
from collections import Counter

words = []
dirname = input('Enter directory name: ').strip()

for one_filename in Path(dirname).glob('*.txt'):
    with one_filename.open() as f:
        print(one_filename.name)
        for one_line in f:
            words += one_line.split()

c = Counter(words)
```

```
for word, frequency in c.most_common(10):
    print(f'{word}: {frequency}')
```

There is, by the way, a more sophisticated way to implement this program: a `Counter` object implements +=, allowing you to add one `Counter` to another. Instead of defining `words` as a list, adding the words from each line to it, and then creating the `Counter` at the end of the program, you can just create a main `Counter` (`words`) once, then create a new `Counter` from the words on each line, and running += on it:

```
from pathlib import Path
from collections import Counter

words = Counter()
dirname = input('Enter directory name: ').strip()

for one_filename in Path(dirname).glob('*.txt'):
    with one_filename.open() as f:
        print(one_filename.name)
        for one_line in f:
            words += Counter(one_line.split())

c = Counter(words)
for word, frequency in words.most_common(10):
    print(f'{word}: {frequency}')
```

The preceding code prints each filename as it is read, and then each of the 10 most common words found in those files.

> **NOTE**
>
> `Counter` objects support both += (adding two values together) and |= (keeping the maximum value). In this case, += will do what we want.

AI Challenge: Specify and Compare

For your GenAI task, give an AI system the specifications for this exercise. Then compare its solution with yours. For example, here's what I gave Gemini:

```
I'm reading an intro Python book, and am on the chapter about the
standard library. Here's an exercise:
```

(I entered the exercise specifications here.)

```
I have already implemented a solution based on these
requirements. Please implement your own. I'll then compare mine
with
yours.
```

Here's what Gemini gave to me:

```
Here is an implementation following the exact steps outlined in
your exercise.

```python
from collections import Counter
from pathlib import Path

# 1. Define an empty list, words.
words = []

# 2. Ask the user to enter a directory name.
dir_name = input("Enter a directory name: ")
dir_path = Path(dir_name)

# 3. Use glob to get the text files (i.e., files ending with
'.txt') in
this directory.
# We join the directory path with the '*.txt' pattern.
text_files = dir_path.glob("*.txt")

# 4. Go through each filename, and if it is a regular file, open
it.
for file_path in text_files:
    if file_path.is_file():
        # Print the filename, so that the user gets a status
update.
        print(file_path.name)

        # 5. Go through each line in the file.
        with open(file_path, "r", encoding="utf-8") as f:
            for line in f:
                # Break the line into a list of words, and add
```
```

```
them to words.
                line_words = line.split()
                words.extend(line_words)

# 6. When the program has gone through all of the text files,
# create a Counter object based on words.
word_counts = Counter(words)

# 7. Use the most_common method to display the 10 most common
words
# in these files, and how many times each appeared.
print("\n--- 10 Most Common Words ---")
for word, count in word_counts.most_common(10):
    print(f"{word}: {count}")

```

A Few Design Notes

* **`extend` vs `append`:** In step 5, `line.split()` returns a
list of strings.
We use `words.extend()` to add each individual word to our main
list,
rather than nesting the whole sublist inside it.
* **`encoding="utf-8"`:** It's always a good habit to explicitly
specify the
encoding when opening text files in Python to avoid unexpected
`UnicodeDecodeError` crashes depending on the operating system's
default locale.

How does this match up against your version? If you did anything
differently — like handling punctuation, stripping whitespace, or
dealing
with case sensitivity — I'd love to see it!

Summary

Python's standard library offers solutions, or at least partial solutions, to a wide variety of problems that you're likely to encounter. The fact that it comes with Python means that you don't need to worry about installing anything extra, either on your development computer or wherever the program will run. In the next chapter, you'll learn about PyPI, the Python

Package Index, with hundreds of thousands of third-party packages you can download and use.

Chapter 13. PyPI

In Chapter 12, you saw how useful and versatile Python's standard library is, and how it saves you time and energy by providing commonly needed data structures and functions. As great as the standard library might be, it doesn't reflect the diverse needs of the Python community. Nor does it aim to do that; every addition to the standard library adds to the Python core development team's workload and increases the size of the Python distribution.

That's where PyPI, the Python Package Index, comes in. PyPI is the central repository for third-party Python packages. It's where you can find packages that might be useful in your program. And because it's sponsored by the Python community, it's where you can publish your own packages, if and when you want to do so.

In this chapter, you'll learn how to navigate PyPI, so that you can find, download, and install packages of interest to you. You'll learn about `pip`, the traditional package-installation program, and `uv`, a newcomer that is taking the Python world by storm. By the end of the chapter, you will have learned to use PyPI effectively, leveraging the Python community's work to improve your software while reducing your workload.

Intro to PyPI

You've created a Python package that you're pretty sure would benefit many other programmers. How can you tell them about it? How can you share it with them?

In the early days of Python, there weren't any great answers to this question. Some developers would post their packages on their own servers, telling others about these contributions on public forums and in private blogs. Eventually, developers created a centralized index of such packages —

PyPI, the Python Package Index. PyPI didn't store the packages themselves, but pointed to where they could be downloaded. Over time, it became clear that individuals' servers couldn't guarantee the stability that the Python community needed, especially for popular packages. PyPI took over the storage of packages, as well as their index. Figure 13-1 shows the PyPI home page.

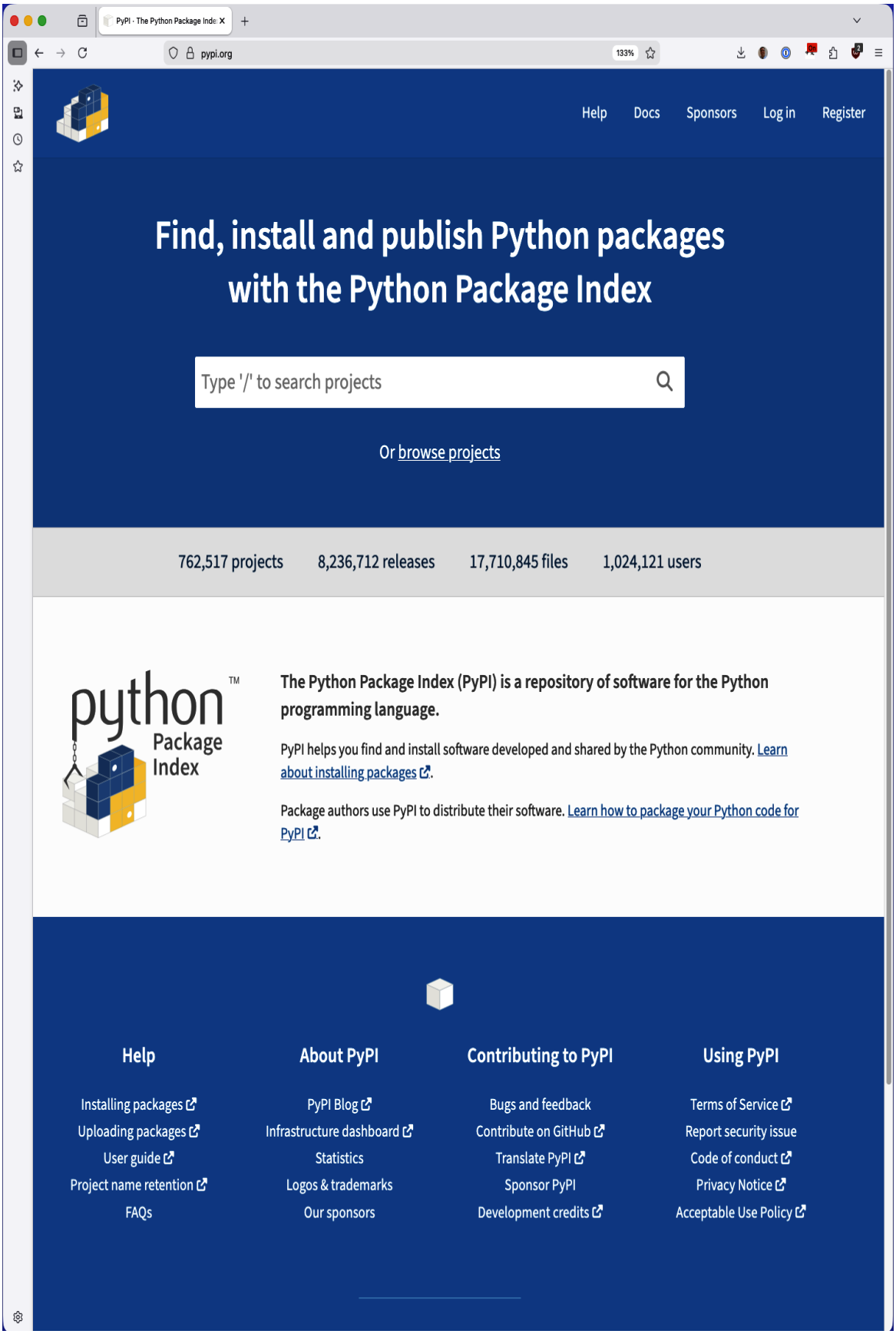

PyPI · The Python Package Inde
pypi.org
133%
Help
Docs
Sponsors
Log in
Register
Find, install and publish Python packages with the Python Package Index
Type '/' to search projects
Or browse projects
762,517 projects
8,236,712 releases
17,710,845 files
1,024,121 users
python Package Index
The Python Package Index (PyPI) is a repository of software for the Python programming language.
PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.
Package authors use PyPI to distribute their software. Learn how to package your Python code for PyPI.
Help
Installing packages
Uploading packages
User guide
Project name retention
FAQs
About PyPI
PyPI Blog
Infrastructure dashboard
Statistics
Logos & trademarks
Our sponsors
Contributing to PyPI
Bugs and feedback
Contribute on GitHub
Translate PyPI
Sponsor PyPI
Development credits
Using PyPI
Terms of Service
Report security issue
Code of conduct
Privacy Notice
Acceptable Use Policy

Figure 13-1. PyPI's home page

Today, PyPI is run by the Python Software Foundation, the nonprofit that manages the Python ecosystem. It hosts, as of this writing, about 800,000 distinct packages, with more than 5 billion packages — or 900 terabytes — downloaded each day. Some of the packages on PyPI are huge, stable, and popular, such as NumPy, Django, Flask, and pandas. Many others are small, abandoned, and nonworking. But they are all part of a community effort to share our work, so that others can avoid reinventing the wheel.

> **NOTE**
>
> PyPI is pronounced "pie pee eye." You might think it should be pronounced "pie pie," and that's a reasonable mistake to make. However, there is an alternative implementation of Python known as PyPy, which the community pronounces, "pie pie."

But wait: why do we even need PyPI? Why not just fold these packages into Python's standard library, and distribute them along with Python? There are at least three reasons:

- Including all of PyPI in the Python distribution would make every download of the Python language unworkably huge. Besides, most PyPI packages are irrelevant to most Python developers; why force everyone to download everything when they can pick and choose?
- The standard library is only distributed with Python. If a standard-library module has a bug, it's only fixed when the next release of Python comes out. (New major releases come out every October, and bug-fix releases come out every few months.) Including all of PyPI's packages in the standard library would force all module authors to be on that schedule, or would force more frequent updates to Python, neither of which is very desirable.
- Rights to the Python standard library, along with the rest of the Python distribution, are owned by the Python Software

Foundation. Not all people want to sign over ownership of their code to the PSF.

While you can thus depend on the standard library always being available, packages from PyPI need to be specified and installed as part of any project you create. You'll see how to do this later in the chapter, when you learn about the `uv` package-management tool.

Using PyPI

PyPI is organized by *project*, where each project contains one or more Python packages. (Each package, you might recall, includes one or more Python modules.) Each project on PyPI has a unique name, and its own page on the site under https://pypi.org/. For example, the "rich" project is at *https://pypi.org/project/rich*, and the "numpy" project is at *https://pypi.org/project/numpy*. Figure 13-2 shows the project page for `rich`.

rich · PyPI

pypi.org/project/rich/

Type '/' to search projects

Help Docs Sponsors Log in Register

rich 14.3.3

pip install rich

✓ Latest version

Released: Feb 19, 2026

Render rich text, tables, progress bars, syntax highlighting, markdown and more to the terminal

Navigation

- Project description
- Release history
- Download files

Verified details

These details have been verified by PyPI

Maintainers

willmcgugan

Unverified details

*These details have **not** been verified by PyPI*

Project links

- Documentation
- Homepage

Meta

- **License:** MIT License (MIT)
- **Author:** Will McGugan
- **Requires:** Python >=3.8.0
- **Provides-Extra:** `jupyter`

Classifiers

Development Status

- 5 - Production/Stable

Project description

python 3.8 | 3.9 | 3.10 | 3.11 | 3.12 | 3.13 | 3.14 pypi package 14.3.3

downloads/month 284M codecov 98% blog rich news Follow @willmcgugan

rich_

English readme • 简体中文 readme • 正體中文 readme • Lengua española readme • Deutsche readme • Läs på svenska • 日本語 readme • 한국어 readme • Français readme • Schwizerdütsch readme • हिन्दी readme • Português brasileiro readme • Italian readme • Русский readme • Indonesian readme • فارسی readme • Türkçe readme • Polskie readme

Rich is a Python library for *rich* text and beautiful formatting in the terminal.

The Rich API makes it easy to add color and style to terminal output. Rich can also render pretty tables, progress bars, markdown, syntax highlighted source code, tracebacks, and more — out of the box.

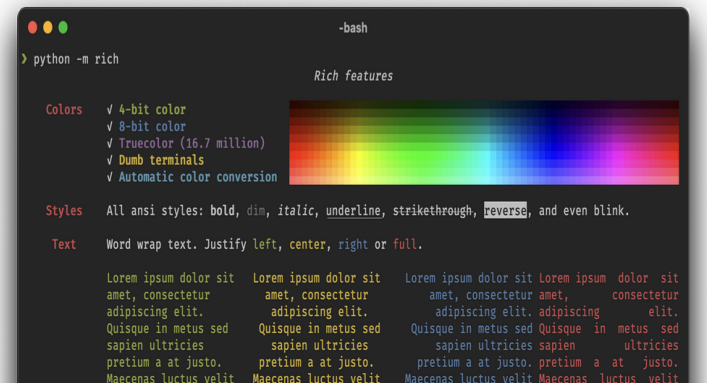


Figure 13-2. PyPI's project page for `rich`

A project page on PyPI typically has several parts:

- An overall project description (shown by default)
- A "release history" link, showing when each version of this project was published to PyPI
- A "download files" link, from which you can download the package to your computer
- A list of the maintainers responsible for the project's PyPI page
- A link to the project's external home page, if it has one
- A link to the project's open-source repository, typically on GitHub
- An indication of what open-source license applies to the project

In the best-case scenario, the main description page will include a complete introduction to the package, some sample code, and hints for getting started.

Want to install a package from PyPI onto your computer? You'll learn more about this later in the chapter, but the easiest and fastest way is with `pip`, a command-line program that comes with Python. You will likely see instructions telling you to write the following in your terminal:

```
pip install rich
```

However, getting the `pip` command to work from your command line can sometimes be frustrating or complex. For that reason, many Python developers suggest that you invoke `pip` within a larger invocation of `python`:

```
python -m pip install rich
```

If all goes well, you'll be able to write and execute the following Python program:

```
import rich
rich.print('[green]Hello![/green] :thumbsup:')
```

In the first line of the preceding program, you `import rich`, using the same syntax as you would for a package from the standard library. You might remember that `import` looks through each of the directories in `sys.path`, and that the final location in its search is `site-packages`, the directory into which `pip` installs packages from PyPI. The interfaces for loading a module from the standard library and PyPI are identical.

Once imported, `rich` behaves like any other module, with numerous functions. One of them is `rich.print`, which allows you to add colorization, formatting, and emoji hints to the string as it is printed.

> **NOTE**
>
> Don't confuse the `rich.print` function, meaning the `print` function defined in the `rich` module, with the `print` function that comes with Python. This is a great example of why the namespace feature of Python modules is so useful; it means that a module can define any functions and variables it wants, without having to worry about collisions with other modules' chosen names.

For many Python programmers, especially when they start out, this is the main way that they work with PyPI and packages:

- They find a package on PyPI.
- They install it with `pip`.
- They `import` it and use it, as if it were a standard-library module.

That might describe you! The rest of this chapter will go into some detail and nuances about `pip` usage, package selection, and installation. But if you're in a hurry, then just find, `pip install`, and `import` might be enough to get you started.

> **NOTE**
>
> If you're using an integrated development environment (IDE) such as PyCharm or VS Code, then you can install packages without ever looking at the command line. That's because these programs allow you to search for, download, and install any package you want from PyPI, right from within the environment.

Downloadable Files

As you saw before, a PyPI project page has a link to "downloadable files." Typically, there will be at least two files on that page:

- A *wheel*, meaning a file with a *.whl* extension. This is the most common way to distribute Python packages. It's basically a ZIP file containing the package's Python code, along with administrative data about the package, such as what packages must also be installed in order for it to be installed. The filename for a wheel indicates not only the package name and version, but also what version of Python and what operating system it's for. If a package has components written in C or other compiled languages, then there might be multiple wheels, each for a specific architecture.
- A *source distribution*, a compressed file (with a *.tar.gz* extension) containing the source code needed to compile and install the package on your own computer. Years ago, this was the typical way in which Python packages were distributed, and there are still sometimes reasons to use them. However, most developers can ignore source distributions, relying on wheels — which are easier, faster, and more reliable to deal with.

If you want, you can download a wheel from the downloads page, onto your own computer. You can then use the `pip` program to install it from the command line:

```
pip install rich-14.3.3-py3-none-any.whl
```

However, there is almost no reason to do this! That's because `pip` will go to PyPI on your behalf, retrieve the latest version of a package that is also appropriate for your platform, and install it into your computer's `site-packages` directory.

As you already saw, the preferred way to install `rich` is:

```
python -m pip install rich
```

The `-m pip` tells Python to load and use `pip`. You can then issue `pip` commands, such as `install`, in this way.

If `rich` is already installed on your computer, then `pip install` won't install a newer version — even if a newer version is available on PyPI. That's to avoid upgrading to a newer version, which could potentially break things. To force `pip` to install a newer version, use the `-U` option:

```
python -m pip install -U rich
```

When you use `pip` to install a package, you aren't just installing a single file onto your computer. The wheel that `pip` downloads will contain some Python code, contained in one or more files. But it'll also include a number of other components, from a project description file to support data.

Moreover, a package can indicate that it depends on the installation of one or more other packages. For example, the `pandas` data-analysis library relies on the `numpy` package, and indicates this in its project file. If you use `pip` to install `pandas`, then `pip` will first check to see if `numpy` is installed. If not, then both will be installed. For this reason, installing one package with `pip` can sometimes lead to a cascade of other installations, to ensure that all of the dependencies are met.

Exercise: Installing a Package

In this exercise, you'll use `pip` to install a PyPI package (`roman-numerals`) onto your computer. You'll then invoke some of its functions.

Specifications

Here are your instructions for this exercise:

- Search for "Roman numerals" on PyPI. How many matches do you find?
- Now use `pip` to download and install the `roman-numerals` package from PyPI.
- Write a program that asks the user to enter two Roman numerals, and prints their sum in Roman numerals.

Example:

```
Enter first Roman numeral: XI
Enter second Roman numeral: IV
XI + IV = XV
```

You can solve this exercise interactively, with the help of an AI tutor.

Solution

First, you need to install the `roman-numerals` package from PyPI:

```
python -m pip install -U roman-numerals
```

With that in place, you can then `import` the package. Note that the package name includes a `-` sign, which isn't legal in Python module and package names, because it cannot be in a Python identifier (i.e., variable or function name). For this reason, the package is actually imported as `roman_numerals`. And the actual name you'll want to use from there is `RomanNumeral`:

```
from roman_numerals import RomanNumeral
```

Next, get the user's input — but as Roman numerals! This means:

- Use `input` to get the user's input
- Run `strip`, to remove any leading or trailing whitespace
- Invoke `RomanNumeral.from_string` on the input, to get a new `RomanNumeral` value from each

Here's how that code looks:

```
from roman_numerals import RomanNumeral

rn1 = RomanNumeral.from_string(input('Enter first Roman numeral:
').strip())
rn2 = RomanNumeral.from_string(input('Enter second Roman numeral:
').strip())
```

How, then, can you sum `rn1` and `rn2`? You need to convert them both to integers, using `int`:

```
from roman_numerals import RomanNumeral
rn1 = RomanNumeral.from_string(input('Enter first Roman numeral:
').strip())

rn2 = RomanNumeral.from_string(input('Enter second Roman numeral:
').strip())

total = int(rn1) + int(rn2)
```

And then, to print the final value as a Roman numeral? You'll need to invoke `RomanNumeral` on total:

```
from roman_numerals import RomanNumeral
rn1 = RomanNumeral.from_string(input('Enter first Roman numeral:
').strip())

rn2 = RomanNumeral.from_string(input('Enter second Roman numeral:
').strip())

total = int(rn1) + int(rn2)
```

```
print(f'{rn1} + {rn2} = {RomanNumeral(total)}')
```

Notice that if you want to create a `RomanNumeral` value from an integer, you can just invoke `RomanNumeral(n)`. But if you want to create it from a string, you need to use the special method `RomanNumeral.from_string('VI')`. That's a bit confusing and surprising — but it is in the documentation, which is good.

AI Challenge: Find the Bug

Ask a GenAI chatbot to take the program that you wrote, and to add a bug. Can you find it?

Here's the output from a ChatGPT session in which I did that:

```
I'm reading an intro Python book, and I'm in the chapter on PyPI.
Here
are the instructions for the latest exercise:

* Search for "Roman numerals" on PyPI. How many matches do you
find?
* Now use `pip` to download and install the `roman-numerals`
package
  from PyPI (https://pypi.org/project/roman-numerals).
* Write a program that asks the user to enter two Roman numerals,
and
  prints their sum in Roman numerals.

Example:

---
Enter first Roman numeral: XI
Enter second Roman numeral: IV
XI + IV = XV
---

My solution:

from roman_numerals import RomanNumeral
rn1 = RomanNumeral.from_string(input('Enter first Roman numeral:
').strip())
```

```
rn2 = RomanNumeral.from_string(input('Enter second Roman numeral:
').strip())

total = int(rn1) + int(rn2)

print(f'{rn1} + {rn2} = {RomanNumeral(total)}')

I want you to help me learn by inserting a bug into my code, and
asking me to find it. Keep going, with new bugs and challenges,
until
I tell you to stop.
```

ChatGPT responded:

```
Great exercise. Here's your code with **one deliberate bug**
inserted:

```python
from roman_numerals import RomanNumeral

rn1 = RomanNumeral.from_string(input('Enter first Roman numeral:
').strip())
rn2 = RomanNumeral.from_string(input('Enter second Roman numeral:
').strip())

total = int(rn1) + int(rn1)

print(f'{rn1} + {rn2} = {RomanNumeral(total)}')
```

Your challenge: run through it carefully and find the bug.

A few things to check:

* Are both user inputs being used correctly?
* Does the addition match the printed expression?
* What result would you get for `XI` and `IV`?

When you think you've found it, tell me what the bug is and how
you'd
fix it, and I'll give you the next one.
```

Finding a Package

PyPI has grown rapidly over the last decade. That growth is both a blessing and a curse: there is almost certainly a package to handle any topic, interface, or problem you're attacking. But finding that package, or choosing among several different candidate packages, isn't so easy. That's because PyPI's user interface can be frustrating to deal with.

For example, PyPI's home page offers a search bar and encourages you to enter the topic you're searching for. If you enter "database", you'll be told that there are more than 10,000 projects matching your search term, as shown in Figure 13-3. In the top right of the page, you can choose whether you want to order them by relevance (i.e., how often your search term appears in the package's documentation) or by when it was last updated. Neither is that helpful.

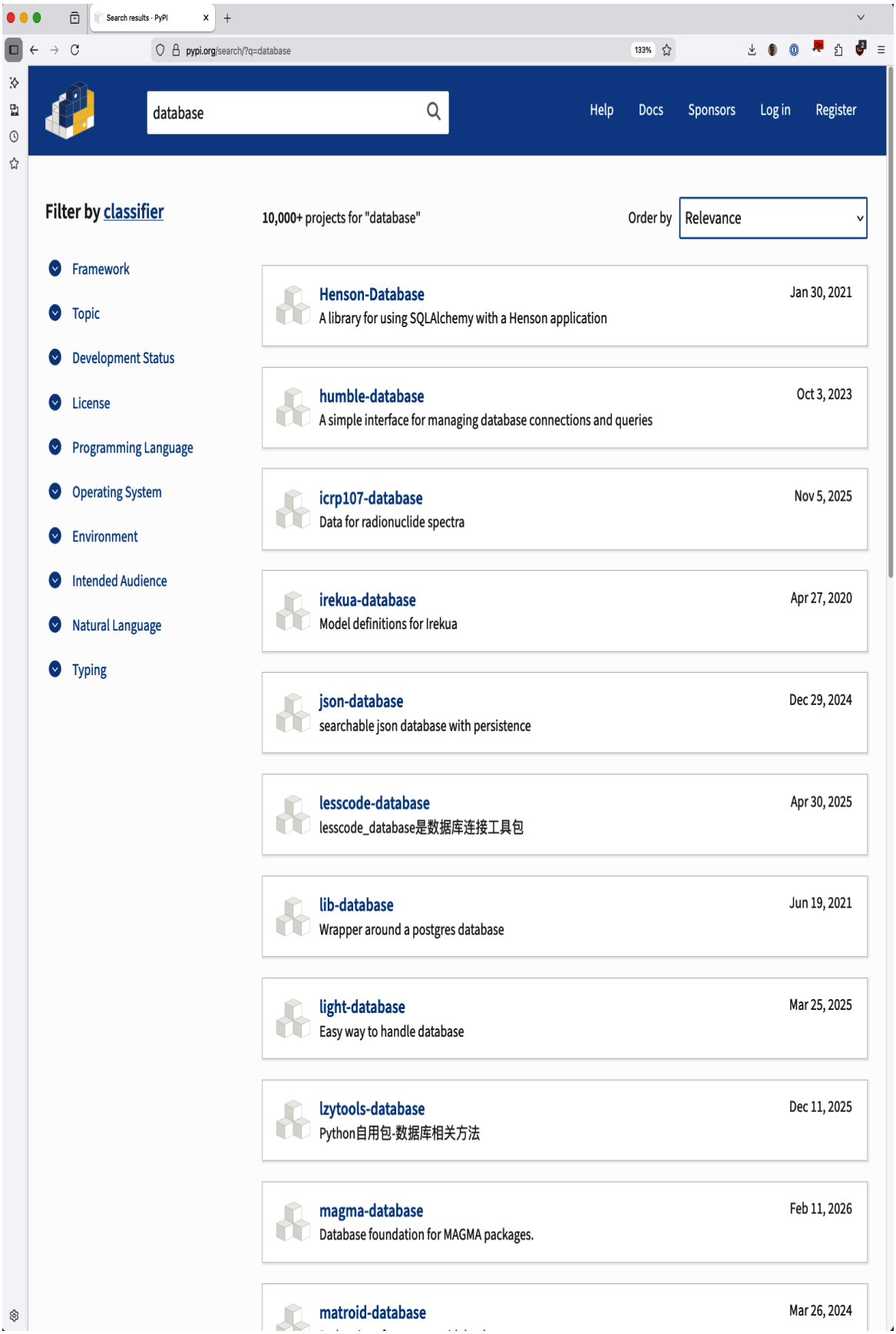

Search results · PyPI
pypi.org/search/?q=database
133%
database
Help
Docs
Sponsors
Log in
Register
Filter by classifier
Framework
Topic
Development Status
License
Programming Language
Operating System
Environment
Intended Audience
Natural Language
Typing
10,000+ projects for "database"
Order by
Relevance
Henson-Database
A library for using SQLAlchemy with a Henson application
Jan 30, 2021
humble-database
A simple interface for managing database connections and queries
Oct 3, 2023
icrp107-database
Data for radionuclide spectra
Nov 5, 2025
irekua-database
Model definitions for Irekua
Apr 27, 2020
json-database
searchable json database with persistence
Dec 29, 2024
lesscode-database
lesscode_database是数据库连接工具包
Apr 30, 2025
lib-database
Wrapper around a postgres database
Jun 19, 2021
light-database
Easy way to handle database
Mar 25, 2025
lzytools-database
Python自用包-数据库相关方法
Dec 11, 2025
magma-database
Database foundation for MAGMA packages.
Feb 11, 2026
matroid-database
Mar 26, 2024

Figure 13-3. Search results for "database"

You can narrow down the search results by choosing one or more *classifiers* on the left side of the screen, as shown in Figure 13-4. But the number of classifiers is also huge, making them quite difficult to use, too.

Figure 13-4 shows what it looks like to filter PyPI packages using classifiers.

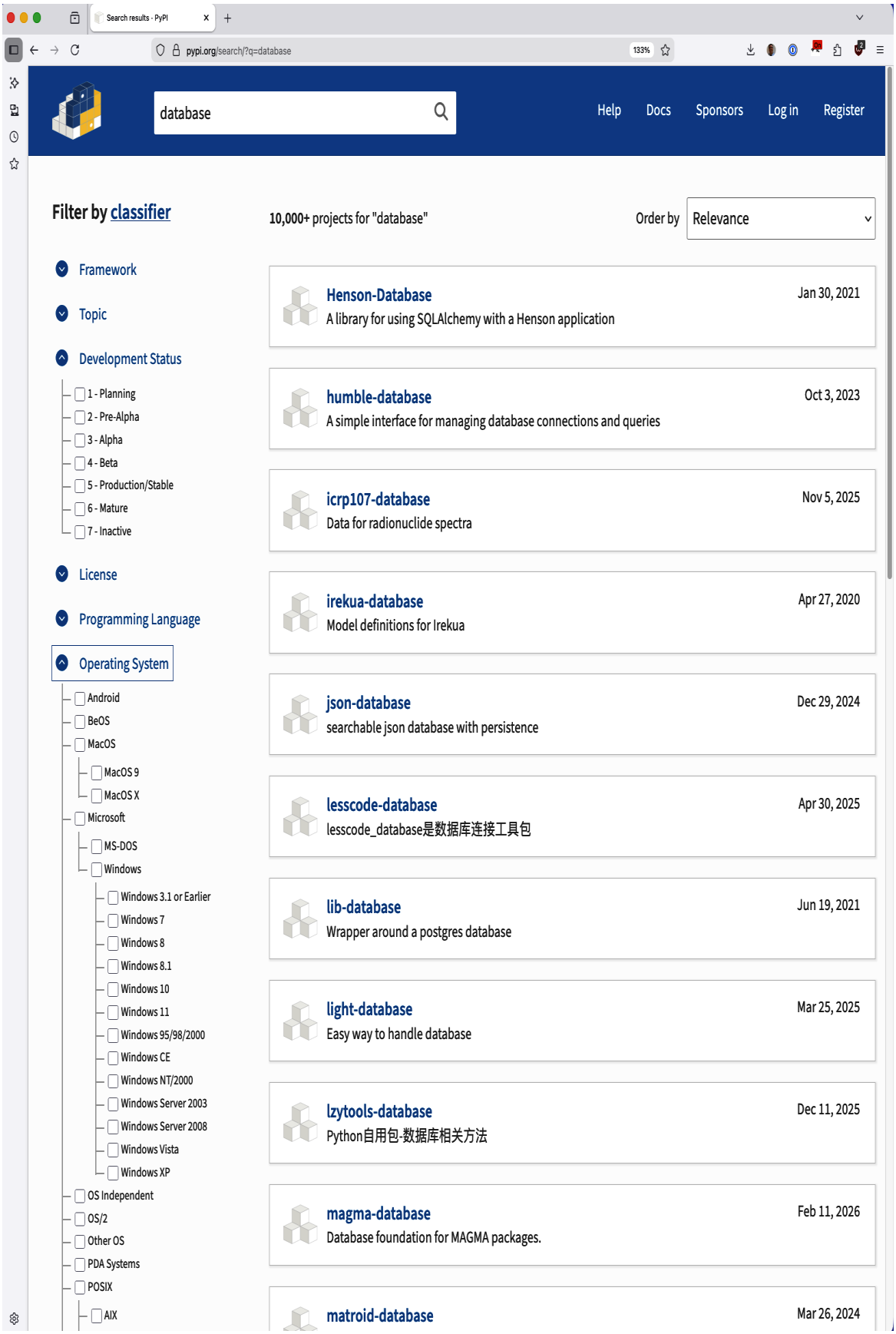

Search results · PyPI
pypi.org/search/?q=database
133%
database
Help
Docs
Sponsors
Log in
Register
Filter by classifier
10,000+ projects for "database"
Order by
Relevance
Framework
Topic
Development Status
1 - Planning
2 - Pre-Alpha
3 - Alpha
4 - Beta
5 - Production/Stable
6 - Mature
7 - Inactive
License
Programming Language
Operating System
Android
BeOS
MacOS
MacOS 9
MacOS X
Microsoft
MS-DOS
Windows
Windows 3.1 or Earlier
Windows 7
Windows 8
Windows 8.1
Windows 10
Windows 11
Windows 95/98/2000
Windows CE
Windows NT/2000
Windows Server 2003
Windows Server 2008
Windows Vista
Windows XP
OS Independent
OS/2
Other OS
PDA Systems
POSIX
AIX
Henson-Database
A library for using SQLAlchemy with a Henson application
Jan 30, 2021
humble-database
A simple interface for managing database connections and queries
Oct 3, 2023
icrp107-database
Data for radionuclide spectra
Nov 5, 2025
irekua-database
Model definitions for Irekua
Apr 27, 2020
json-database
searchable json database with persistence
Dec 29, 2024
lesscode-database
lesscode_database是数据库连接工具包
Apr 30, 2025
lib-database
Wrapper around a postgres database
Jun 19, 2021
light-database
Easy way to handle database
Mar 25, 2025
lzytools-database
Python自用包-数据库相关方法
Dec 11, 2025
magma-database
Database foundation for MAGMA packages.
Feb 11, 2026
matroid-database
Mar 26, 2024

Figure 13-4. PyPI classifiers

In some ways, the best way to find good PyPI projects is to find them outside of PyPI. For example, people often discuss interesting PyPI packages at conference talks and user-group meetings, as well as in podcasts, newsletters, and YouTube videos. If you hear someone speak about an interesting-sounding package, it might be worth checking out. Their endorsement is typically worth more than a random encounter.

Another good resource for finding good PyPI packages is Awesome Python. This is a list of Python packages that are known to be useful and maintained. Listings here tend to point to the package's home page on GitHub, rather than to the project page on PyPI. However, it's a rare package that doesn't direct you to the PyPI project page, or tell you how to install it with `pip`.

If you're looking to solve a problem, and don't really know where to start, you should probably begin with Awesome Python (shown in Figure 13-5), rather than searching on PyPI.

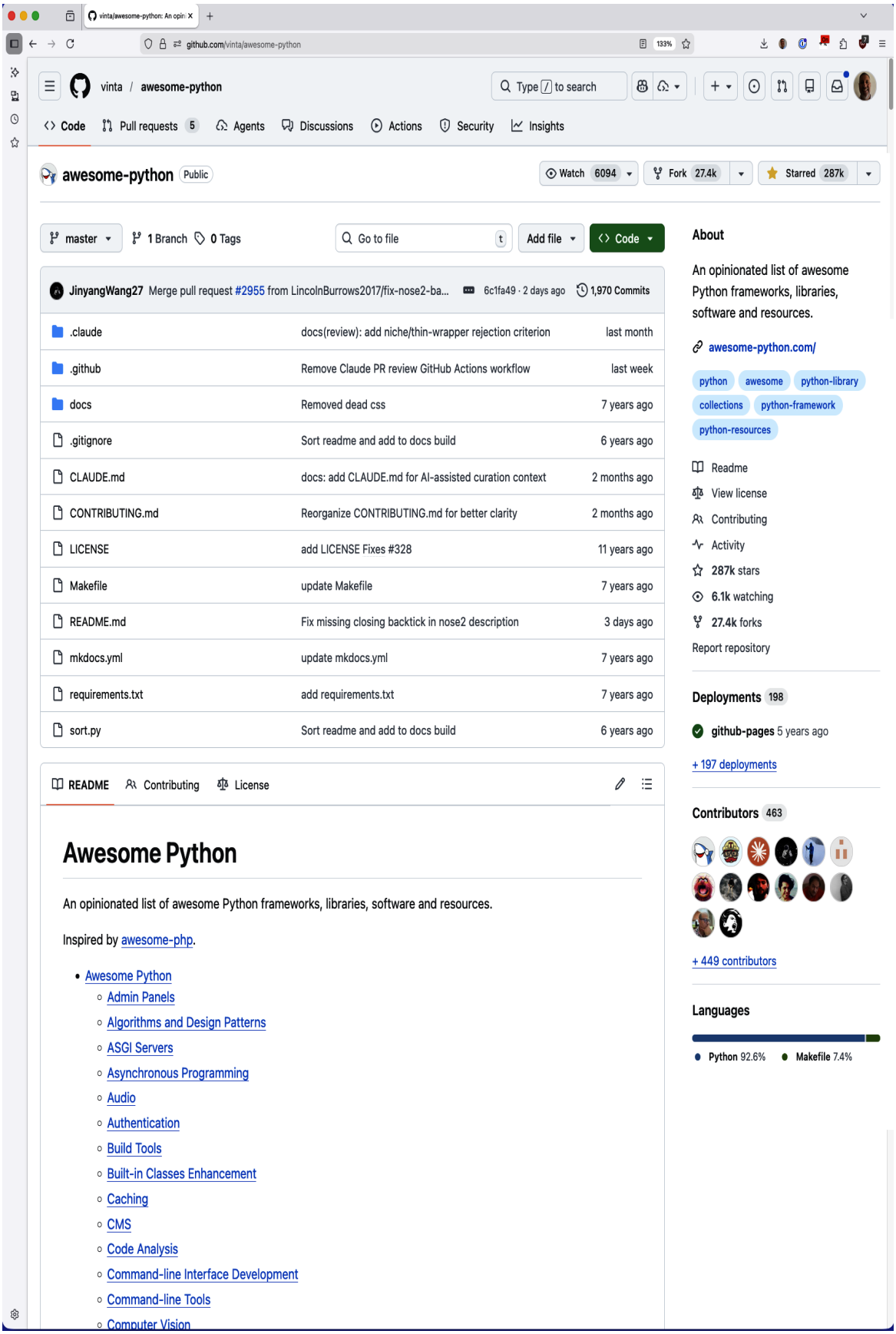

vinta/awesome-python: An opini
github.com/vinta/awesome-python
133%
vinta / awesome-python
Type / to search
Code
Pull requests 5
Agents
Discussions
Actions
Security
Insights
awesome-python Public
Watch 6094
Fork 27.4k
Starred 287k
master
1 Branch
0 Tags
Go to file
t
Add file
Code
JinyangWang27 Merge pull request #2955 from LincolnBurrows2017/fix-nose2-ba... 6c1fa49 · 2 days ago 1,970 Commits
.claude docs(review): add niche/thin-wrapper rejection criterion last month
.github Remove Claude PR review GitHub Actions workflow last week
docs Removed dead css 7 years ago
.gitignore Sort readme and add to docs build 6 years ago
CLAUDE.md docs: add CLAUDE.md for AI-assisted curation context 2 months ago
CONTRIBUTING.md Reorganize CONTRIBUTING.md for better clarity 2 months ago
LICENSE add LICENSE Fixes #328 11 years ago
Makefile update Makefile 7 years ago
README.md Fix missing closing backtick in nose2 description 3 days ago
mkdocs.yml update mkdocs.yml 7 years ago
requirements.txt add requirements.txt 7 years ago
sort.py Sort readme and add to docs build 6 years ago
README
Contributing
License
Awesome Python
An opinionated list of awesome Python frameworks, libraries, software and resources.
Inspired by awesome-php.
Awesome Python
Admin Panels
Algorithms and Design Patterns
ASGI Servers
Asynchronous Programming
Audio
Authentication
Build Tools
Built-in Classes Enhancement
Caching
CMS
Code Analysis
Command-line Interface Development
Command-line Tools
Computer Vision
About
An opinionated list of awesome Python frameworks, libraries, software and resources.
awesome-python.com/
python
awesome
python-library
collections
python-framework
python-resources
Readme
View license
Contributing
Activity
287k stars
6.1k watching
27.4k forks
Report repository
Deployments 198
github-pages 5 years ago
+ 197 deployments
Contributors 463
+ 449 contributors
Languages
Python 92.6%
Makefile 7.4%

Figure 13-5. Awesome Python

Evaluating a Package

Trying to find a package among the hundreds of thousands of options can feel like looking for a needle in a haystack, because nearly any search will bring up an overwhelming number of possible matches.

Whether it's from a search, or a recommendation you saw online, or an experienced Python developer who recommended it, assume that you have found what seems like the perfect package. You want to install it, and then `import` it into your program.

Should you?

If you're wondering why this could be an issue, consider that PyPI is meant to let Python coders share packages with one another. This basically means that anyone with an internet connection can upload a package to PyPI. Which, in turn, means that anyone on the internet can write code that will then be executed on other users' computers. As you can imagine, this raises many questions.

Quality and Safety

First, is a package any good? This is, in many ways, the hardest question to answer. PyPI doesn't have a rating system, so you can't look for five-star packages. But you can typically go to the home page for the project's source code, often hosted on GitHub, and examine a few things there:

- How many GitHub stars (i.e., endorsements from the open-source community) does a package have?
- How many contributors does a package have?
- When was the package last updated?
- How good is its documentation?

If a package has several hundred contributors, several thousand commits (including some in the last few days), and thousands of GitHub stars, then it's likely to be of high quality. By contrast, if it has a handful of commits by a single person, no GitHub stars, and hasn't been updated in a few years, then it's almost certainly not a good idea to download and use it, even if it seems to solve your problem.

Most packages fall between these two extremes, meaning that there isn't a single, clear indication of whether a package is good or not. You'll need to do some investigation, and some consideration, when making a decision.

A second question to ask is: is a package potentially dangerous? The Python Software Foundation, which runs PyPI, tries hard to balance between keeping the service open, inviting anyone and everyone to contribute, while recognizing that there are bad actors who want to distribute malicious code via PyPI. There are full-time PSF staff members who work to automate scans and checks of the code on PyPI, and to date, actual problems have been relatively rare. But it doesn't mean that security problems are unknown, or that the current checks will always suffice.

Fortunately, the heuristics for avoiding security problems are similar to those for finding high-quality packages: the odds of a long-standing package, with a large user base and developer community, containing malicious code are fairly slim. But they aren't zero, and if you're ever concerned about the trustworthiness of a package, then you should avoid it yourself and even report it to PyPI's security team.

Licensing

It's also important to consider the software license of any PyPI package you use. That might seem odd, given that everything on PyPI is open source. But while many people think "open source" is a synonym for "free of charge," the term really describes a family of software licenses, each with slightly different requirements. Some licenses, like the MIT, Berkeley, Python, and Apache licenses, basically say, "Take this code, and do whatever you want with it." Most organizations don't seem to have trouble

with such code, or your use of it in your projects. These open-source licenses allow, and even encourage, their use in commercial software.

But some open-source code, including packages on PyPI, comes under the GNU Public License, or "GPL." The GPL says, more or less: take the code, and do whatever you want with it. On the surface, that sounds the same as the licenses mentioned in the preceding paragraph. However, there is one catch: whoever receives the code, or anything using it, has the same privileges and rights as you did.

Meaning, if your software includes an MIT-licensed package from PyPI, then you have the option of distributing your package as open-source or commercial software. But if your software includes a GPL-licensed library, then what you write is effectively open source; even if you declare it to be commercial and closed, the fact that it used a GPL library means that your customers have the right to receive the source code, modify it, and distribute it to others.

Things get even more complex and far-reaching if a program includes libraries licensed under the GPL's "affero" license, which means that even if the software is accessed over a network (as in a web application or API server), users have the right to receive, modify, and redistribute the software's source code.

This doesn't mean that you should avoid GPL-licensed packages on PyPI. But it does mean that you can and should pay attention to the license. And if you're writing Python code as part of your job at a commercial company, you should be careful about downloading and using GPL-licensed libraries. Many companies now have open-source usage policies, describing which licenses are acceptable for different corporate use cases.

Using uv

When you start to use Python, packaging seems simple and straightforward: most things can be done with modules from the standard library. And if the functionality you want isn't there? There's probably something on PyPI that

does the trick. Just download and install the package with `pip`, then `import` it into your Python program.

But as you start to use Python for more tasks, and more projects, you will discover that there are problems with Python's packaging system. Indeed, Python conferences would almost always have at least one talk explaining how packaging worked, and how to avoid the pitfalls.

These problems stem from the fact that a given Python installation has a single `site-packages` directory, and that only one version of a package can be installed at a time. This means that if two projects use different versions of the same package, or conflicting packages, then it might not be possible to install and use both of them on the same computer. If you're working on two projects and one requires `rich` version 1.0 but the other requires `rich` version 2.0, you're kind of stuck.

The traditional solution has been *virtual environments*, in which each project has its own private */site-packages/* directory. Virtual environments do work, but they're confusing and frustrating.

Also, if you want to install your project on another computer, or even collaborate with other engineers on different computers or platforms, how can you guarantee that everyone is using the same packages and package versions? You have to ensure that the packages you're using in your project are compatible with one another. Different projects might need to use different versions of Python. And what if you want to publish your project to PyPI?

There were always solutions to these problems. But each solution required a different piece of software, and often multiple, competing pieces of software.

And then, a few years ago, came `uv`, and changed everything. You can think of `uv` as a Python packaging super-app, doing everything you might need:

- It creates and manages Python projects, using best practices that keep each project siloed and separate.

- It handles the downloading and installation of packages.
- It checks for compatible (and conflicting) packages on every architecture and platform, to reduce conflicts.
- It installs and manages Python versions.
- It builds wheels.
- It publishes your package, if you want, to PyPI.

On top of all this, `uv` is lightning fast, often hundreds of times faster than `pip`.

> **NOTE**
>
> How is `uv` so ridiculously fast? Among other things, it's written in Rust, a compiled language that has the memory and speed advantages of C, but many fewer downsides and complications. This leads people to claim that `uv` is a super-fast `pip` replacement. And that is true, but `uv` is so much more than that. If you're using its `pip` compatibility mode, using commands like `uv pip install`, then you aren't using `uv` to its fullest.

In this section, you'll learn about how to use `uv`. You'll soon see why it has taken the Python world by storm, and might decide to use it in your own projects.

Installation and Basic Use

`uv` is from Astral, a company that makes a number of other open source Python tools — most famously `ruff`, which checks and reformats your code. (Astral was acquired by OpenAI in the spring of 2026.) Instructions for downloading and installing `uv` are available on their website. Once installed, use `uv` as your primary command, rather than `python` or `pip`.

Every command consists of `uv` *`something`*, where *`something`* is what you actually want to do.

Using `uv` means working inside *projects*. Each project is basically a directory or folder on your computer in which you'll write your code. A project might contain a single Python file, and it might contain hundreds. You can have as many projects as you want on a given computer.

For example, you can create a project /myproj/ with:

```
uv init myproj
```

This creates a new subdirectory, `myproj`, beneath the current directory. It is there that you can create your Python program files, Jupyter notebooks, configuration files, and anything else you want or need. You'll typically want to switch into that directory (using `cd` on the command line) when issuing `uv` commands.

On your screen, you'll see something similar to this:

```
Initialized project `myproj` at `/Users/reuven/Desktop/myproj`
```

If your project contains a program, *main.py*, then you can switch into the project directory and run it (from within the directory) with `uv run`:

```
cd myproj
uv run main.py
```

Using `uv run` ensures that you're using the version of Python that `uv` has installed and is configured to use, rather than whatever version was installed by your operating system and happens to be in your terminal's search path.

When you create a project with `uv init`, `uv` sets up the project directory with a number of files and subdirectories:

- It creates the *pyproject.toml* file that is the current standard for describing and specifying Python projects. In particular, as you'll soon learn, this file indicates which third-party packages need to be installed.

- It turns the project directory into a Git project under version control, albeit without any commits. If you use Git — and you probably should! — then you'll welcome the fact that Git is already set up for you.
- It creates a *.gitignore* file, telling Git which common Python programs, files, and directories should be ignored.
- It creates a *.python-version* file, indicating what version of Python should be used when executing programs in this project.

Here is the default *.gitignore* file:

```
# Python-generated files
__pycache__/
*.py[oc]
build/
dist/
wheels/
*.egg-info

# Virtual environments
.venv
```

In other words, Git shouldn't store cached Python modules (`__pycache__`), compiled Python files (**.pyc* and **.pyo*), or the contents of the *build*, *dist*, and *wheels* directories. It should also ignore any files from the old-style "egg" distribution format. And of course, *.venv*, the name of the directory into which a virtual environment will be created, is also ignored.

pyproject.toml

Python is used in a wide variety of domains, and in large applications that go beyond one Python file — or even multiple Python files. For this reason, the Python community has adopted the notion of a *project*. When you use `uv init`, you're creating such a project.

Each project has a configuration file, called *pyproject.toml*. This tells a variety of Python tools (including `uv`) about your project, including what packages it requires and what version of Python it should use. The file is written in a format known as TOML, short for Tom's Obvious, Minimal Language, which is an easy-to-read, easy-to-write format for configuration files.

Here is the *pyproject.toml* file that `uv` created for *myproj*:

```
[project]
name = "myproj"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.14"
dependencies = []
```

From this, you can see:

- Everything, by default, is within the `[project]` section.
- The project has a version number, starting with "0.1.0".
- The project has a description, which you can and should update from this default.
- The project has a *README.md* file, which you can and should update.
- This project will work with any Python version starting with `3.14`.
- This project doesn't require any third-party packages from PyPI.

To learn more about *pyproject.toml*, what it does, and how to modify it, see the official documentation.

You might have noticed that the Python version is specified twice, once in *pyproject.toml* and a second in `.python-version`, a file created by `uv`. These have slightly different roles to play:

- The `requires-python` entry in *pyproject.toml* describes the limits for the project. When Python 3.15 comes out, this *pyproject.toml* file will be just fine using that version.
- The `python-version` file, by contrast, says what version should be used right now, on this computer, when you invoke `uv run`. So if `requires-python` is `>=3.14`, but `.python-version` contains the version `3.14`, then even if both 3.14 and 3.15 are installed and available, 3.14 will continue to be used.

Managing Packages

When you invoke a program with `uv run`, `uv` runs it within a *virtual environment*. This has nothing to do with virtual machines or cloud computing; it simply means that Python will ignore the global *site-packages* directory on your computer, and only look in a project-specific *site-packages* directory. If every project has its own *site-packages*, then there isn't any way to encounter the problem you learned about earlier, namely that two projects on your computer fight over which package version should be installed.

For virtual environments to work, two things need to happen:

- First, Python needs to look in your virtual environment's *site-packages* directory, rather than the global one. Running Python via `uv run` guarantees that this will indeed happen.
- Second, when you install packages from PyPI, they should be put into the project-specific *site-packages*, rather than in the shared, global directory.

For things to be installed in the right place, you'll want to replace `pip install` with `uv add`. When you run `uv add`, it installs the package into a *site-packages* directory within your project. In that way, it doesn't interfere with the *site-packages* directories for any other projects you've created.

> **NOTE**
>
> `uv run` doesn't just look in your project's *site-packages* before the global *site-packages*. It looks there *instead* of the global *site-packages*. If you run a program via `uv run` and try to `import` a package that was installed globally, Python will not see it. That's because the global *site-packages* isn't in the Python variable `sys.path`.

If your program needs `rich`, then you can download and install it into your project with:

```
uv add rich
```

This does several things:

- It updates the *pyproject.toml* configuration file, adding `rich` to its list of required packages.
- It installs `rich` into the project-specific *site-packages* directory.

Once you are using `uv`, you shouldn't ever use `pip` again. Just use `uv add` to add a package from PyPI to your project. Want to upgrade a package? Just use `uv add --upgrade` and the package you want to upgrade.

Why, if `uv` is so great, does this book still tell you about `pip`? Because `pip` remains the standard way to install and manage packages, comes with Python, and is still in use in many organizations. But if you can switch to `uv` from `pip`, you'll likely be very happy with the result.

Using uv and IDEs

If you're using a modern IDE such as PyCharm or VS Code, then you don't have to use `uv` from the command line. These programs allow you to create a new project that is managed by `uv`. Figure 13-6 shows the startup screen for a new PyCharm project.

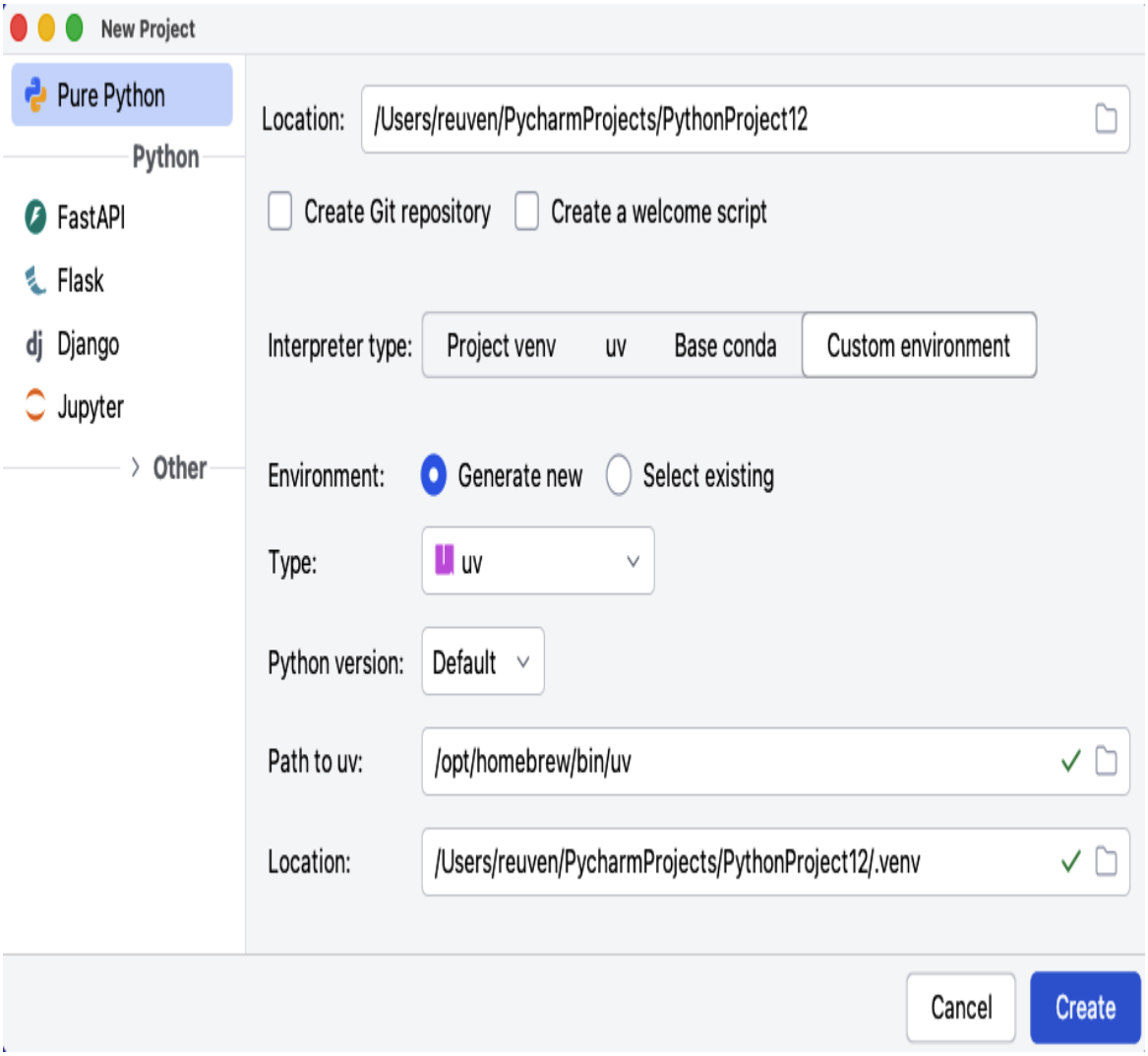


Figure 13-6. Create a new PyCharm project

Note that when creating a new PyCharm project via `uv`, the *pyproject.toml* file is used — but the directory is not initialized as a Git repository. Neither the `.git` subdirectory nor the `.gitignore` file will be created.

Figure 13-7 shows a new project just after being created with `uv` in PyCharm.

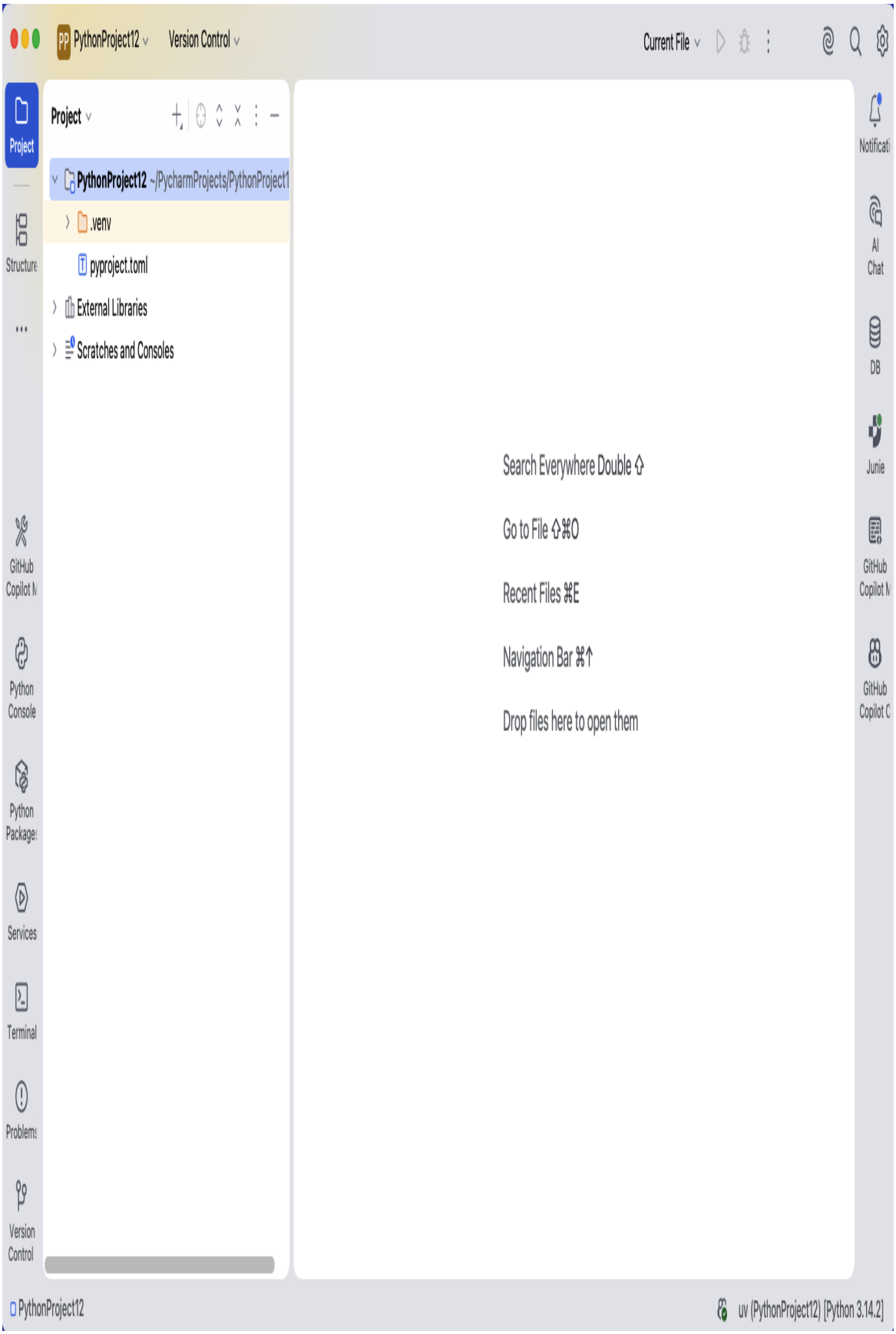

PythonProject12
Version Control
Current File
Project
PythonProject12 ~/PycharmProjects/PythonProject1
.venv
pyproject.toml
External Libraries
Scratches and Consoles
Search Everywhere Double ⇧
Go to File ⇧⌘O
Recent Files ⌘E
Navigation Bar ⌘↑
Drop files here to open them
Structure
AI Chat
DB
Junie
GitHub Copilot
Python Console
Python Packages
Services
Terminal
Problems
Version Control
PythonProject12
uv (PythonProject12) [Python 3.14.2]

Figure 13-7. New PyCharm project with `uv`

You can then write any program you want. However, if you `import` a module that hasn't been added, you'll see a red squiggly line under the module's name, indicating it needs to be installed (see Figure 13-8).

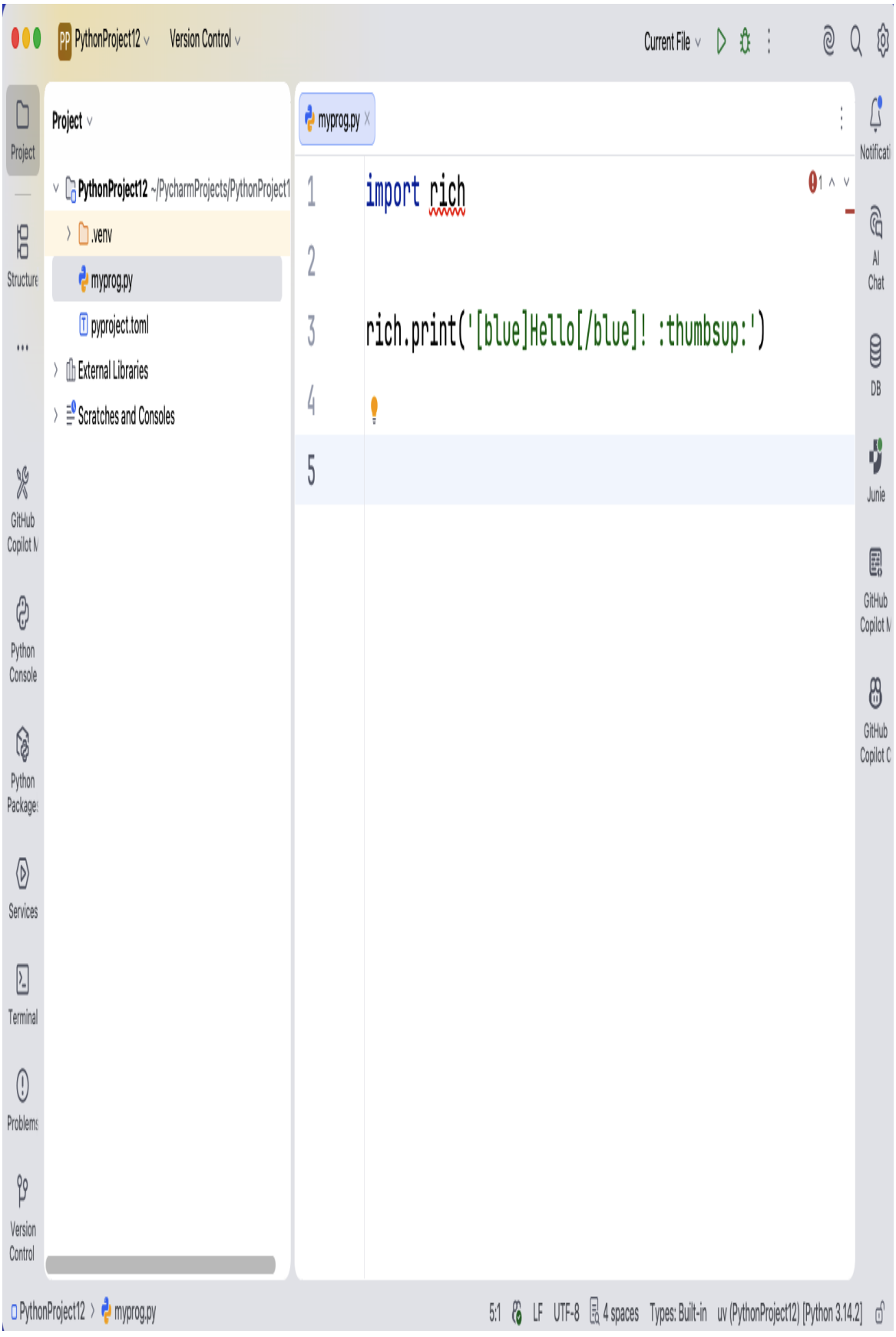

PythonProject12
Version Control
Current File
Project
PythonProject12 ~/PycharmProjects/PythonProject1
.venv
myprog.py
pyproject.toml
External Libraries
Scratches and Consoles
myprog.py
1 import rich
2
3 rich.print('[blue]Hello[/blue]! :thumbsup:')
4
5
Structure
GitHub Copilot M
Python Console
Python Packages
Services
Terminal
Problems
Version Control
Notificati
AI Chat
DB
Junie
GitHub Copilot M
GitHub Copilot C
PythonProject12 › myprog.py
5:1 LF UTF-8 4 spaces Types: Built-in uv (PythonProject12) [Python 3.14.2]

Figure 13-8. PyCharm project trying to use `rich`

Hovering over the name of the imported (but not installed) module, `rich`, offers the option to install it. Clicking on that link adds the package — and because the project is managed via `uv`, it gets added to *pyproject.toml*, as shown in Figure 13-9.

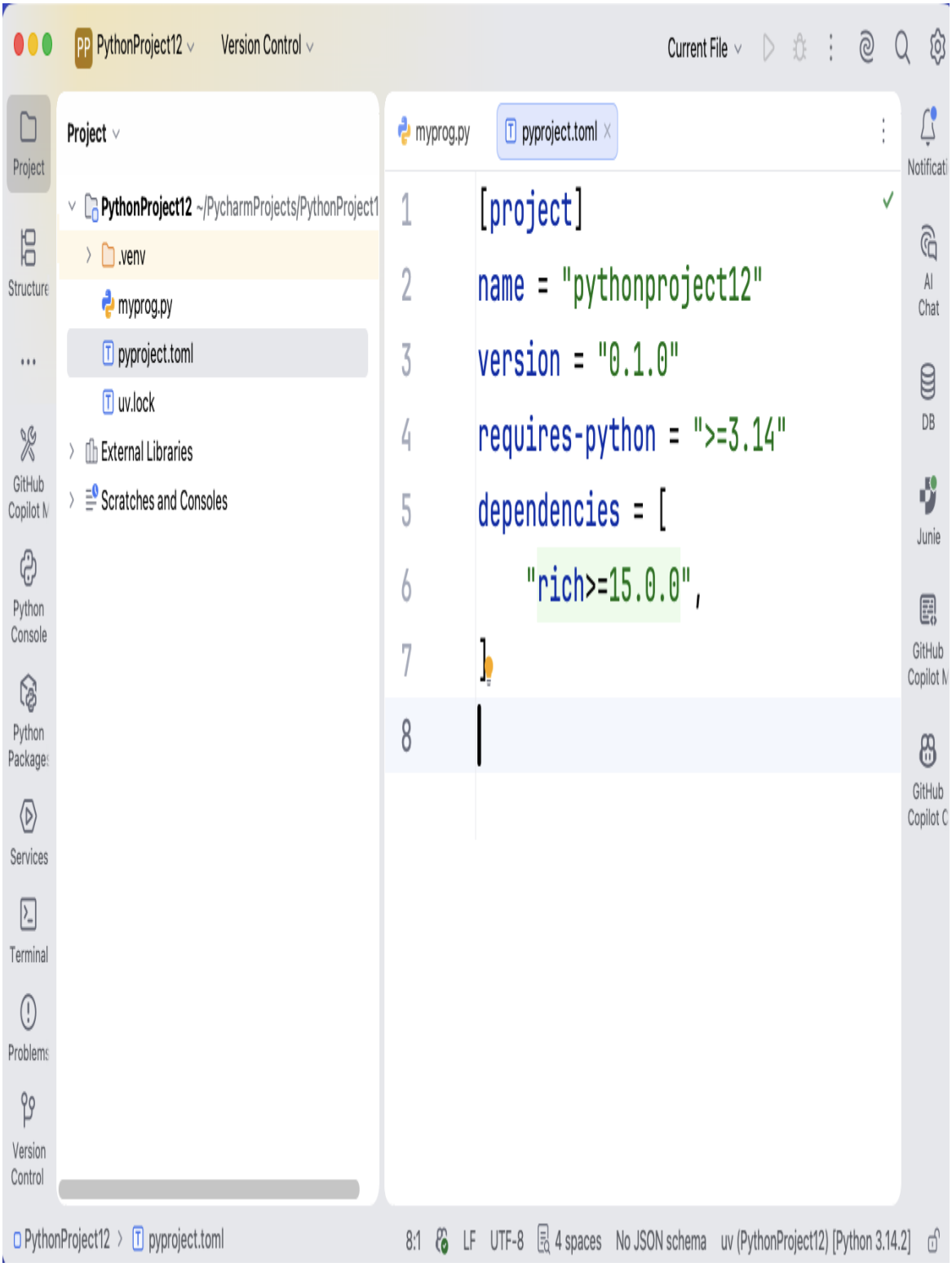


Figure 13-9. pyproject.toml file after adding `rich` via `uv`

Exercise: Using rich

In this exercise, you will create a `uv` project, and then write a short program that uses the `rich` module to print some text in color.

Specifications

Here's what you need to do:

- Install `uv` on your computer.
- Use `uv init` to create a new project, `userich`.
- Edit the *main.py* file inside of `userich`. It should `import rich` and use `rich.print` to display some colorized text. (For documentation and examples, refer to the PyPI page about `rich`.)
- Run the program with `uv run`. See that the program doesn't work, exiting with an error from trying to `import rich`.
- Use `uv add` to add the `rich` package from PyPI to `userich`.
- Run *main.py* again, and see that it does work.

You can solve this exercise interactively, with the help of an AI tutor.

Solution

To create a new project, use `uv init`:

```
uv init userich
```

Following that, you can go into the project directory and edit the *main.py* file. By default, that file looks like this:

```
def main():
    print("Hello from userich!")
```

```
if __name__ == "__main__":
    main()
```

You can change it to say:

```
import rich

def main():
    rich.print("[blue]Hello[/blue] [yellow]from[/yellow]
    [green]userich[/green]!")


if __name__ == "__main__":
    main()
```

You can then run it, via the command line, with:

```
uv run main.py
```

However, because `rich` has not been installed into this project, you'll get an error:

```
Using CPython 3.14.2
Creating virtual environment at: .venv
Traceback (most recent call last):
  File "/Users/reuven/Desktop/userich/main.py", line 1, in
    import rich
ModuleNotFoundError: No module named 'rich'
```

The solution?

```
uv add rich
```

This produces some brief output:

```
Resolved 5 packages in 313ms
Installed 4 packages in 36ms
 + markdown-it-py==4.0.0
 + mdurl==0.1.2
 + pygments==2.19.2
 + rich==14.3.3
```

It also modifies the *pyproject.toml* configuration file to indicate that `rich` is required for this project to work:

```
[project]
name = "userich"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.14"
dependencies = [
    "rich>=14.3.3",
]
```

With this in place, you can again try to run the program:

```
uv run main.py
```

Sure enough, it now prints its output in three colors.

AI Challenge: Specify and Compare

How would a GenAI system solve this problem? Go to your favorite chatbot, and give it the same specifications that you got. (If you can do it in a coding environment that works inside your terminal, such as Claude Code, that would be even better.) What does it create? And how does its output compare with your solution?

Summary

PyPI contains countless treasures — packages that you can use in your programs, allowing you to save time and benefit from work done by others in the Python community. Knowing how to evaluate, download, install, and use these packages in your own programs, whether with `pip` or `uv`, is a crucial part of being a productive Python developer.

You have now learned the basics of how to code in Python. You have learned the basic data structures, how to organize your code into functions,

how to use the modules that come with Python, and how to download and install third-party packages from PyPI. In Chapter 14, you'll see how you can use modern AI tools to assist you when writing your code.

Chapter 14. AI-Assisted Coding

The next time you get together with friends, ask everyone to stand in a circle, holding hands. Then ask two people to each put one hand on a terminal of a battery. Everyone in the circle will get an electric shock. Fun for everyone, right?

> **WARNING**
>
> I hate that I have to say such a thing, when it should be obvious, but…please don't do this.

Where did I get such a strange (and extremely dangerous) idea? From none other than Benjamin Franklin, an American founding father and one of the most famous scientists of the 18th century. In Franklin's day, electricity was at the cutting edge of science. People didn't understand what it was, and certainly didn't have any practical applications for it. But they did store electricity in a primitive battery known as a *Leyden jar*, and were delighted to find that the electric shock passed through people and certain materials.

It's hard for us, in the modern era, to think of electricity as new, exciting, and impractical. But Franklin lived decades before the invention of the telegraph, and about a century before the invention of the electric light bulb. He and his friends couldn't imagine that electricity would eventually power cities, vehicles, computers, and communication.

Why mention this? Because after decades of research, AI is finally starting to demonstrate practical use cases and benefits. We're just starting to emerge from the Leyden jar era of AI, and beginning to see how and where to use it. However, we're still in the first stages of practical AI use. It's amusing and helpful — but many people are already asking whether and when we'll start to see measurable productivity gains from our use of AI.

On the other hand, AI is already able to write some code, and it stands to reason that it'll be able to code as well as humans within a few years. Indeed, a number of leading programmers are saying that by the end of 2026, most code will be written by AI.

Needless to say, this raises a whole host of big questions, including:

- Will people still be needed to program computers?
- Can't we just code using English (or another human language), rather than a specific programming language?
- What computer skills should people be learning?

Beyond these more general questions, there's also the specific question of what tools to use and how to use them. This is complicated by the fact that AI tools are changing extremely fast.

This is the first of three chapters that address coding Python with AI, rather than the Python language itself:

- In this chapter, you'll learn about the current state of AI, a bit about how it works, and about some AI tools you can use to assist you in coding. In these examples, the AI is acting as your sparring and debate partner, allowing you to get feedback in real time. You'll write Python code, but it'll be in partnership with AI.
- In Chapter 15, you'll learn about agent-based coding. This is a very new approach, one in which you don't write the code directly, but rather ask AI to write it for you. Needless to say, this is not only new but highly controversial, for a variety of reasons. You'll learn what agent-based coding is, what it isn't, how you should approach it, and the arguments for and against this approach.
- In Chapter 16, you'll hear what I believe are the skills people need now, and will continue to need in the future.

A Brief History of AI

In the 1960s, leading computer-science researchers thought that it would be possible to create a computer that thought like a human. They coined the phrase *artificial intelligence* to describe this goal. If you've ever been late with a project at work, then these pioneering AI researchers should help you feel better; they estimated that it would take only a few weeks to achieve their goal.

In the following decades, AI researchers pushed ahead in a number of directions. They developed new algorithms, including *neural networks*, simulations of a human nervous system. They explored what it meant for a computer to make decisions, to understand natural (human) language, and even to express itself in language. They developed techniques that would allow a computer to understand human speech, and also to produce speech. There were successes, but even after many decades of work, the notion of having a computer think and work like a human still seemed very remote.

However, research continued — and while it wasn't the only technique people were exploring, a branch of AI known as *machine learning* was of particular interest. Traditional AI research focused on defining rules that would allow a computer to reason like a human. When an AI system would make a mistake, the creators would either change the existing rules or add new ones.

Machine learning took a different approach, feeding inputs (known as *training data*) to a program (known as a *model*), and then asking the model itself to determine the rules based on those inputs. Once a model has been trained, you can ask it for a prediction. For example, let's say you train a model on weather in your part of the world, where hot days are also dry. You can then ask the model to predict whether it'll rain given the temperature. It won't always be right, but it'll often be pretty close, assuming that the future is similar to the past. You can improve a model's accuracy by giving it more training data. You can also ask it to pay attention to more factors, known as *features*, when it is being trained.

Machine learning was already gaining steam as an approach to artificial intelligence, when several different things happened:

- *Deep learning*, a machine-learning technique based on large, complex, and deep neural networks, proved itself extremely capable in language- and image-related tasks.
- The internet provided a nearly infinite body of documents and images for training models.
- Cloud computing made it possible to parallelize tasks by spinning up thousands of computers simultaneously, making it possible to train models with enormous amounts of data.
- Fast and power-efficient graphics processing chips, originally designed for video-game graphics, were particularly appropriate for machine-learning models.

The next step was to use the machine-learning models' predictions to *generate* new text and images, rather than just analyze what had already been created. The autocomplete feature on your phone is a small-scale version of this, guessing your next word based on your personal history of writing and a more general history that it collects from the population at large.

And then, just a few years ago, OpenAI — a company initially started to explore the limits of artificial intelligence — released ChatGPT. ChatGPT is an example of a large language model, or LLM, wrapped inside a web application that allows you to converse with it. It wasn't the first chatbot, but it was the most general-purpose, fully fledged chatbot available to the entire world. Because ChatGPT was trained on much of the internet, it was able not only to hold conversations, but also translate text, answer questions, summarize text, and even write code.

ChatGPT unleashed an explosion of investment, interest, and use of AI. Other companies created competitors to ChatGPT, including Google (with Gemini) and Anthropic (with Claude). Each company trains its model in different ways, and puts different limits on the types of output it can

produce. You can use these companies' models for free, but you can typically get higher-quality output and more usage if you use the paid version, typically costing between $10 and $20 each month. There are also free, open-source models that you can download and use on your own computer — which might not be at the cutting edge of AI models, but give you more flexibility, both in terms of cost and execution speed. That said, such models do often require a great deal of memory and computing power to be competitive with the commercial ones.

It's important to remember that while AI models seem intelligent, and are certainly useful, they aren't intelligent. They are pattern-matching machines, using the text that you have entered and returning text that is statistically likely, given the training inputs. But models have numerous limits: they don't know things beyond their training data, typically meaning that they don't know about current events. They cannot create something truly new, although they can create novel combinations of information that already existed.

The language you use to ask questions of an AI chatbot can thus matter a great deal: if you use English, then an LLM will match your query to its extensive training in English, and produce a result. But if you use a more esoteric, lesser-known language, then the LLM will have less knowledge and training, and will likely give you a worse answer — or just a false one.

AI models can and do hallucinate, producing text that is grammatically correct and seems to make sense, but which is factually incorrect. There are numerous examples of such hallucinations on the internet, and some of them can be quite amusing. But just as you shouldn't drive into a river just because your GPS tells you that there's really a road in front of you, you shouldn't believe something just because an AI model tells you that it's true.

Indeed, one of the most important skills you should develop when working with modern AI systems is to argue with them. If an AI chatbot tells you that something is true, ask for proof. Ask follow-up questions. Use your experience and knowledge to push back on the chatbot, to ensure that you're getting accurate results.

AI and Coding

As soon as ChatGPT was released, people were delighted to discover that it could not only answer questions and write essays, but also write code. One of the programming languages in which it was particularly skilled at coding was Python.

That's not a surprise, given that the quality of a machine-learning model's output depends on the quality and quantity of its training inputs. Python has been around for more than three decades, is extensively documented, and is used in many open source projects, whose code is publicly available on the internet. You could say that Python has become popular because it's easy for people to learn and use — and as a result of that popularity, it's also AI's go-to language.

This means that you can ask AI to write Python code, and it'll be able to do so. When ChatGPT was first released, its Python coding skills were good, but far from exceptional. Over time, however, all of the LLMs have improved their coding ability. As you'll see in Chapter 15, writing code via AI is no longer foolish or fictional. That said, the quality of AI-written code generally declines with larger, more complex code bases.

This state of affairs, and the regular improvements we see in AI-based coding, have led to two different things happening at the same time:

- Programmers are embracing AI, which often makes them more efficient.
- Programmers are terrified that their companies will see how efficient programmers are with AI, and will lay off many of them.

If programmers are indeed far more efficient when using AI, then it stands to reason companies might need fewer programmers. But it's also possible that companies will decide that they want to take on larger, more ambitious projects, and will need each of their programmers to do more than they have done before.

Needless to say, the lack of clarity is leading to confusion, frustration, and nervousness among many coders.

However things shake out in the coming years, there's no doubt that AI tools are already playing a part in many programmers' daily routines. We can expect their importance will only grow over time. What sorts of functionality do these tools offer, and how can you integrate them into your work? Moreover, how *should* you integrate them into your work? The remainder of this chapter will introduce you to the various forms of AI assistance you can use today.

Pair Programming with Chatbots

Throughout this book, you have been using GenAI chatbots in a variety of ways:

- To strategize about how to solve problems,
- To give you step-by-step feedback as you write your code,
- To check over your code, once you have written it, and
- To find bugs in code that doesn't work.

Together, these constitute a technique known as *pair programming*, in which two people share one computer (including one keyboard) and work together to solve problems. There are many variations on this setup, including remote pairing, where the two coders are connected online. In many pairing configurations, the person who is writing the code isn't the one actually typing at the keyboard, but is dictating.

To many, pair programming — and especially, disconnecting the programmer from typing into an editor — seems like a recipe for frustrating, slow coding. But in fact, pairing often produces more readable, maintainable code with fewer bugs. There are a number of reasons for this:

- When you have to communicate your ideas to someone else, you're forced to think through things more carefully.

- Your pairing partner, upon hearing your plans, has a chance to consider and push back on them before writing the actual code. This forced discussion means that by the time it is actually written, it has gone through more careful consideration; you haven't just written the first thing that comes to mind.
- You have two pairs of eyes looking at the code as it is written, and watching the output. If something looks odd, then one of the pairing partners is likely to notice it.
- If (when) there are bugs or problems, it's easier for two people to identify, debug, and correct them. Moreover, the correction has to go through the two-person articulation and discussion loop that happened in the first place.

So yes, the code is written more slowly. But each part of it has gone through multiple rounds of discussion and thought. If and when there are problems, they can be handled almost immediately by the people who wrote the code, and thus know it best.

Besides, experienced programmers know that it's always easiest to fix design problems and eliminate bugs early on in the development process. Even if the initial coding takes a bit longer, pairing can be more than worthwhile if the results are better thought out and more robust.

There are companies that always have their coders work in pairs. And given the aforementioned benefits, it seems clear why they would do so. However, most programmers don't pair all of the time. This is sometimes because egos get in the way. Programmers don't only have a "can-do" attitude toward their work, but also an "I'll do it myself" attitude, as well. Some programmers believe that if the software-writing process is more difficult and frustrating, then the output will be of higher quality.

But in many cases, programmers don't pair simply because of the logistics involved. Coders, like other employees at large companies, often spend half of their time in meetings, leaving less than half of each day for actual

programming work. Coordinating your programming time with a colleague can be difficult, or just inconvenient.

Many programmers have solved this problem doing what's known as *rubber duck programming*. If a key advantage of pairing is the forced communication, then forcing yourself to say your plans out loud before you actually do them is a good idea, regardless of whether someone else is listening. Programmers have thus taken to putting a rubber duck (or another doll or toy) on their desk, telling the duck about their plans before they execute them. This isn't as good as actual pair programming, but it does provide an opportunity to think things through before committing them.

AI changes this, giving you something that's more responsive than a rubber duck. (Whether it's more responsive than a person depends on the person, and also depends on whether you're measuring quality or quantity.) And indeed, you've seen how using an AI chatbot — entering your plans before and as you're coding — can force you to think through what you're doing.

You can also ask a chatbot for its opinion about what you're planning, or about which of two options might be better. For example, if you aren't sure whether it would be better to use a list or a dictionary to store an address book, you can ask the chatbot. The more information you provide, including what data you want to store, how you want to retrieve it, and how many records you'll be storing, the better the response you'll get.

As always, the most important part of using a chatbot is to argue with it. If you simply accept its judgment, you will inevitably make some terrible decisions. And just as your teacher never wanted to hear that the dog ate your homework, your manager isn't interested in hearing that problems in your code are the fault of AI. When a chatbot presents you with a solution or suggestion, ask follow-up questions. Ask for it to justify what it has told you. Ask if there might be a better way to solve the problem. You'd be surprised just how often chatbots respond to such pushback with, "You're right, I hadn't considered everything," followed by a superior solution.

Another advantage of a chatbot window is precisely that it's disconnected from your coding editor. You're planning, not coding, which means that

you're still in the mindset of planning, rather than trying to finish debugging a problem or fixing a feature.

There's nothing wrong with using a chatbot in this way. Indeed, many people started using chatbots like this, and continue to do so today, with great effectiveness. However, the technology has improved — and in the following sections, you'll learn how AI can be integrated into your development editor. This turns out not only to be more convenient, but also provides more functionality than you can get in a chatbot window.

IDEs and AI

So far, this book has encouraged you to use a notebook environment, such as Jupyter, for your coding. But most Python programmers don't write their code in notebooks. Rather, they use an editor, storing their code in files. Modern editors are often known as *integrated development environments*, or *IDEs*, because they integrate the editor (used to write code) with a debugger (used to find and solve problems) and other tools. For example, most modern IDEs work on the basis of a Python *project*, something you learned about in Chapter 11. A project typically consists of a directory, a *pyproject.toml* file describing the project and its dependencies, and then one or more Python program files.

Most developers have strong opinions about their editor. They won't ever change, because it's so central to the way that they work…until they try another one, and switch to it. It's not unusual for a developer to change editors every few years. However, there are some people who have used the same editor for decades, arguing that they cannot possibly be as productive with a different one.

If you haven't tried an IDE yet, then you should definitely give one or more a whirl. Two of the most popular IDEs in the Python world are PyCharm (from JetBrains) and Visual Studio Code (from Microsoft). Both are free, although PyCharm has a paid ("professional") version that includes more features.

Depending on where you work, you might have complete freedom to choose an IDE or none at all. Some companies standardize on a particular IDE, and demand that all employees use it. Others allow employees to choose from among several options. And yet others, often small startups, allow each developer to choose whatever editor feels right to them.

It used to be that a good IDE was one with a good debugger and a sophisticated system for indenting and colorizing code. Today, IDEs manage packages, check your code's style, perform some type checking, and interface with version-control systems, among other things. It's thus no surprise that modern IDEs are also the console via which programmers can communicate with AI systems.

Indeed, the rest of this chapter will look at GitHub Copilot (available in all IDEs), and how you can interact with it from inside your Python program. You'll see that having the AI system inside your IDE takes pair programming to a completely new level, whether you are simply asking for advice and feedback, or (as you'll read in Chapter 15) it's writing the code for you.

Note that there is a new crop of AI-first editors that aim to replace PyCharm and VS Code by putting AI functionality front and center — not just for answering questions about code and helping to complete function definitions, but actually writing large quantities of code. Two of the most famous examples of such editors are Cursor and Windsurf, but there are many others. You'll learn a bit more about these in Chapter 15, but the point of this section of the book is to help you understand and feel comfortable with the use of AI when coding, regardless of the editor or tool you choose to use.

GitHub Copilot

What happens if you modify your code, and discover that instead of fixing a bug, you actually made things worse? This is a common state of affairs in the software world. To avoid this unpleasant situation, programmers have used *version-control systems* for decades, saving backup versions of code

on a regular basis. If something goes wrong in version 2.5, you can revert to version 2.4 or even 2.0, back when things were working.

One of the most popular version-control systems in use today is known as Git. It was first developed by Linus Torvalds, the inventor of the Linux operating system, and quickly became a favorite among many open-source projects. A number of companies were founded to provide developers with storage and collaboration facilities for their Git projects — most famously, GitHub, which was purchased by Microsoft back in 2018. GitHub rose to prominence in no small part because it offered free hosting and storage to open-source projects. Over time, these developers became familiar with Git and GitHub, and started to lobby for their companies to switch to GitHub for their version-control and collaboration needs.

Remember how modern AI systems need to be trained before they can make predictions, and that the better the training data, the better the predictions? In 2021, GitHub released Copilot, a machine-learning system largely trained on its storehouse of source code. (And yes, you read that correctly: Copilot came out about a year before ChatGPT's release.) Programmers could use Copilot to autocomplete programs, rewrite code to be more readable, efficient, or secure, or even to implement pieces of code based on comments.

Copilot, unlike many modern chatbots, was designed to be used within a programming editor. That is: you would fire up PyCharm or VS Code and Copilot was there, ready to offer advice and suggestions. This was a bonanza for many programmers, who could avoid writing repeated, boilerplate code and could concentrate on writing code unique or special for their project.

While Copilot is a paid product ($10/month, as of this writing), there is a free tier that includes a small number of requests each month. You'll first need to get a (free) account on GitHub, at GitHub.com. Then you can sign up for the free Copilot tier.

Copilot and PyCharm

Are you a PyCharm user? If so, it's important to recognize that PyCharm supports at least three different AI systems:

- GitHub Copilot, about which you'll learn in this section
- AI Assistant, an alternative to Copilot
- Junie, an agent-based coding system about which you'll learn more in Chapter 15.

JetBrains wants to let every programmer choose the AI system that best suits them, and thus offers access to Copilot (paid, from GitHub) and their own AI Assistant and Junie products (included in the paid PyCharm Professional subscription). It's nice that they're being flexible, but this flexibility can be quite confusing. Moreover, each of these products has expanded its scope, meaning that the clearly delimited lines in the preceding list are increasingly hazy.

If you have already installed PyCharm — either the free community edition or the paid professional edition — then you can install Copilot by selecting JetBrains from the Download menu on the Copilot home page. That'll take you to the JetBrains plugin site's page for Copilot. Click the Install button at the top of the page, and the Copilot plugin will be installed into your PyCharm. You'll need to provide your Copilot account information so that it can connect you.

Copilot and VS Code

Because VS Code is a Microsoft product, and GitHub is owned by Microsoft, the integration between the two is more straightforward than in PyCharm. Just click on the Copilot icon in the status bar, and sign into GitHub. (If you don't have a GitHub account, you can get one for free.) If you don't have a Copilot account, then you'll be signed up for a free account, with its limitations. You can learn more at VS Code's Copilot setup page.

Autocomplete with Copilot

Sending a text message to a friend? The odds are that you aren't writing every word yourself, but that your phone is doing some of the work, guessing what word you're likely to write next. Instead of writing that word yourself, you can just select it from a menu of options.

Copilot works similarly: once you have it installed, start typing. Once Copilot has enough information to do so, it'll add text, in dimmed text, ahead of your cursor's position. If you want to accept what Copilot has suggested, just press `TAB`. That'll accept the text, and move the cursor past it. You can then edit the text that was just accepted, or go on with additional coding, which Copilot will also try to complete. If you don't like what Copilot is suggesting, then keep typing. It'll keep suggesting based on what it sees.

Just as your phone's autocomplete offers several options for a possible next word, Copilot often offers multiple suggestions for what code it might insert. You can rotate through these different options with Option+`[` and Option+`]` on a Mac, or Alt+`[` and Alt+`]` on Windows and Linux. From my experience, Copilot generally limits itself to only two or three options, and once you get to the end of the suggestions, Copilot will return to the beginning and repeat the cycle.

What if you want to just code, without Copilot suggesting things while you type? VS Code offers a "snooze" button that turns off Copilot's suggestions for five minutes at a time. PyCharm doesn't have such a function, but you can always turn Copilot off completely if you find it distracting.

Let's say you want to write a function that takes a list of integers and returns their sum — the `mysum` function that you saw in Chapter 10. You can say:

```
def mysum(
```

at this point, Copilot springs into action, suggesting the code shown in Figure 14-1:

```
def mysum(a, b):
    return a + b
```

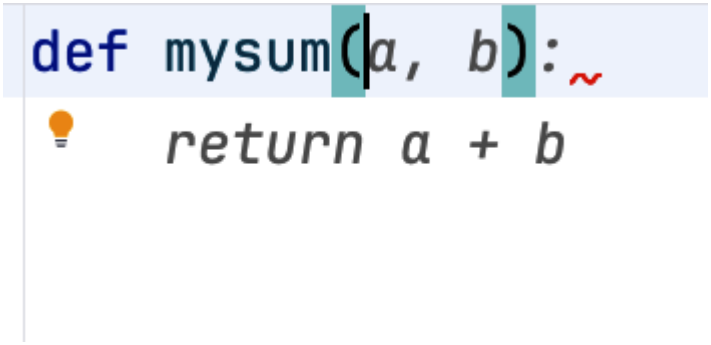


Figure 14-1. First attempted completion of `mysum`

Now, this isn't wrong. It's a perfectly legitimate Python function. But it's not what you had in mind. Remember, Copilot will make a guess based on its training data — and in the majority of its training data, a function named `mysum` takes two arguments and returns their sum. Moreover, Copilot doesn't offer any other suggestions.

However, if you type just a little more, adding the letters `nu` as a parameter, Copilot is likely to change its tune, giving the totally different suggestion shown in Figure 14-2:

```
def mysum(number):
    if number == 0:
        return 0
    else:
        return number + mysum(number - 1)
```

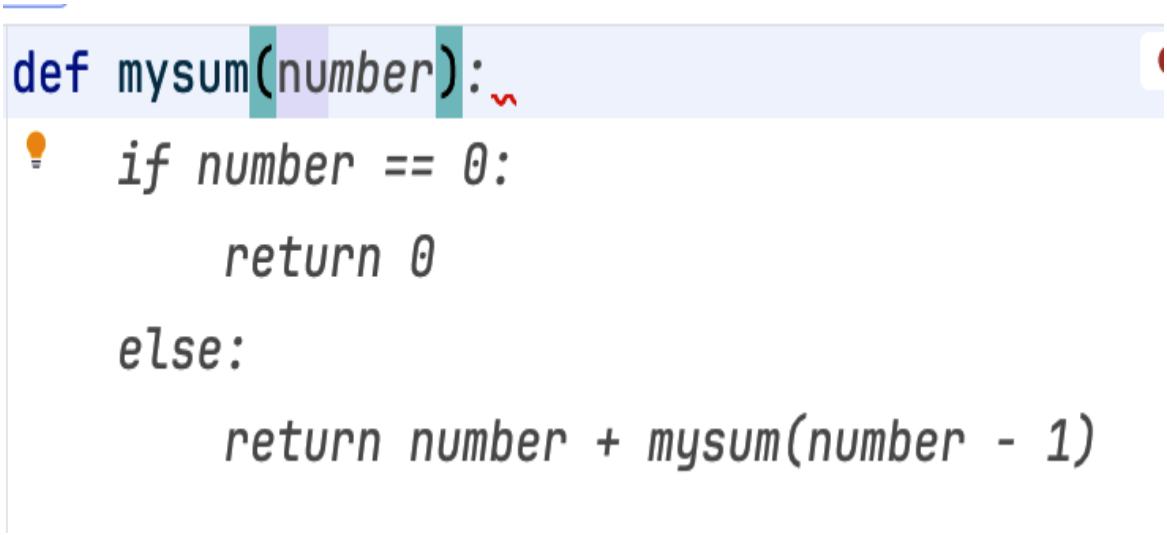


Figure 14-2. Second attempted completion of `mysum`

This is a weird (but legitimate) way to sum numbers, but it assumes that the user is passing a single integer, not a list of integers. That's obvious when you see that the parameter name Copilot provided, `number`, is singular, rather than plural. The singular name, rather than a plural one, indicated Copilot's intentions.

Continuing to type a bit more, such that the parameter name is the plural `numbers`, brings up the solution that you might have expected, shown in Figure 14-3:

```
def mysum(numbers):
    total = 0
    for num in numbers:
        total += num
    return total
```

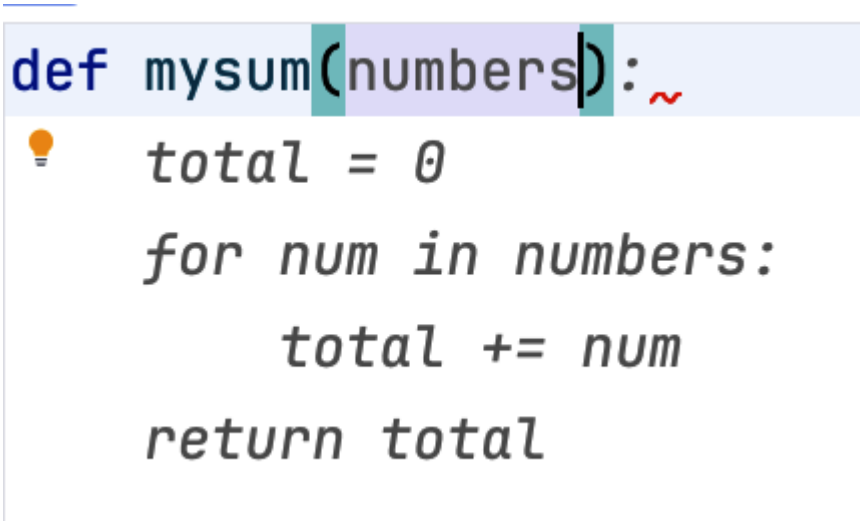


Figure 14-3. Third attempted completion of `mysum`

As you can see, this kind of push-pull with Copilot can save you a great deal of time. However, simply accepting Copilot's suggestions without thinking or checking more deeply can result in code that does something very different from your intentions. It's best to treat Copilot's work as you would code from a colleague with whom you're pair programming: taking it seriously but considering it carefully before simply accepting it.

Chatting with Copilot

Copilot's autocomplete features are powerful and clever. But in many ways, that's just the tip of the iceberg. That's because when you use Copilot in your IDE, you can chat with it about your code. There are numerous ways to use this:

Talk to the chatbot

> When you are first planning your code, you can run your ideas by the chatbot. This is similar to the "strategy session" task that you have done several times in this book. The difference is that Copilot sees your file — so if you're looking to write a new function, it can take the existing ones into account when giving you advice.

Ask questions

Wondering why Copilot chose to implement a function in a particular way? Did it use an unfamiliar method? Do you think that another technique might be better? Then don't hesitate: ask Copilot your questions. The more you ask, the better you'll understand your code, the implications for the way it was implemented, and what alternatives exist. Often, you'll even find that if you ask Copilot why it chose a particular implementation, it'll back down and admit that it should have used a different technique, or that its implementation didn't really work.

Analyze the code

For example, you could ask Copilot, "As a tough, experienced QA engineer, what problems do you see with the code in this file? How could it be improved?" LLMs don't have feelings or egos to protect, and when faced with a prompt like this one, they will give you an honest — brutally honest, even — response. This kind of code review will help you to improve the code that you write, but it'll even improve the code that Copilot writes. (You would be surprised how often an LLM writes code, then gives that same code scathing reviews minutes later.)

Improve the style

You can ask Copilot why it wrote code in a certain way — for example, you could ask why it assigned the result of an operation to a variable, and then returned that variable, when it could have just returned the value right away. It'll usually give you a number of reasons why it did things the way it did, and if appropriate, some reasons why the implementation could have been done in a different way.

Refactor

Refactoring is the term for changing a function's implementation while keeping its inputs (parameters) and outputs (return values) unchanged. Perhaps you've found a more elegant solution to a problem. Or you need to fix a potential security hole. Or you can improve the execution speed, or use less memory. With Copilot, you can highlight the code in

question and ask it for suggestions about how to improve things — or even to refactor the code itself.

Debug

Is your program not working the way you expected? Did you get puzzling output? Again, Copilot has access to the file, and to all of its contents. Ask it for help in debugging. True, the IDE's actual debugger is more powerful, but you would be surprised by how many problems are easy for Copilot to identify, explain, and even fix.

Some of this functionality is built into Copilot with *slash commands*. That is: you can give Copilot specific instructions by typing `/` followed by a command. For example, if you want to better understand the `mysum` function, you can type `/explain mysum` in the chatbot window. Copilot will walk you through the code, line by line. If something still doesn't make sense, you can (and should) ask it follow-up questions. Two additional slash commands are `/fix`, which tries to fix problems in your code, and `/simplify`, which refactors code.

Exercise: Palindromes

In this exercise, you'll write a function that asks a user to enter a word or sentence. The function will return `True` if the text is a palindrome — that is, reads the same forward and backward — and `False` otherwise. You'll then use Copilot to better understand the code that was written.

Specifications

Here's what you should do for this exercise:

- Write a function called `is_a_palindrome`, which takes a single string argument. The string can contain any characters, but only letters and numbers will be included in the check to see if it's a

palindrome; others will be ignored. The comparison will be case-insensitive, so `'ABba'` will be considered a palindrome.

- Use Copilot to help you write this function. If it writes code that is unfamiliar to you, use the chat feature to ask it what it meant.

Solution

This exercise wasn't really about writing code so much as using Copilot to help you write code. As a result, Copilot does a lot of the heavy lifting here — although knowing how to invoke Copilot, and get it to do what you want, requires knowledge and experience.

To write the code, open a new file in an IDE and write:

```
def is_a_palindrome(
```

Almost immediately, Copilot will suggest the following code.

> **NOTE**
>
> Remember that AI, as currently implemented, gives a different answer each time you ask a question, even if it's the same question. So any time that this chapter tells you what AI *will* say, understand that it's *likely* to say this, but that nothing is guaranteed.

Pressing `TAB` accepts the code as is:

```
def is_a_palindrome(word):
    cleaned_word = ''.join(char.lower() for char in word if
char.isalnum())
    return cleaned_word == cleaned_word[::-1]
```

The good news? This code is a fine implementation of the function.

The bad news? It uses two techniques that you didn't learn about in this book. This is precisely when you want to dive into things with Copilot, asking about the syntax, and what it means. Doing this each time you

encounter unfamiliar code is a great way to improve your Python skills. For example, you can enter

```
/explain is_a_palindrome
```

Copilot will then respond with a full, line-by-line explanation:

```
- Explain what is_a_palindrome does in plain terms
- Walk through each step of the logic with short code excerpts
- Clarify why this approach is useful and what inputs it handles
well

You're looking at a helper that checks whether a string reads the
same
forward and backward, while ignoring case and non-alphanumeric
characters. That means punctuation, spaces, and symbols are
removed
before comparison. The function's core setup is:

def is_a_palindrome(word):

So it expects one input (word) and returns a Boolean (True or
False).

The first important step is normalization. This line builds a
cleaned
version of the input:

cleaned_word = ''.join(char.lower() for char in word if
char.isalnum())

It does three things at once: iterates through each character,
keeps
only letters/digits (char.isalnum()), and lowercases what remains
(char.lower()). The result is a canonical string that can be
fairly
checked for palindrome behavior. For example, "A man, a plan, a
canal:
Panama!" becomes "amanaplanacanalpanama".

The actual palindrome check is then very compact:

return cleaned_word == cleaned_word[::-1]

cleaned_word[::-1] is Python slicing syntax for reversing the
```

```
string. If the cleaned string equals its reversed version, the
function returns True; otherwise False. This is an idiomatic and
efficient approach for typical string sizes, and it keeps the
function
concise and easy to reason about.
```

This explanation isn't wrong, but if you're unfamiliar with Python's *comprehensions* (and this book didn't introduce them, since they're somewhat advanced), you still might not understand the normalization and cleaning that takes place. So, you can ask a follow-up question:

```
I still don't understand the line that assigns to cleaned_word,
what
you called normalization and cleaning. What is that syntax? How
does
it work? Please go through it slowly.
```

Sure enough, it answers:

```
- Clarify exactly what cleaned_word = ... is syntactically
- Break it into small pieces (inside → outside)
- Show an equivalent multiline version so the shorthand makes
sense
- Walk through one concrete example character by character

Let's zoom in on this line from myprog.py:

cleaned_word = ''.join(char.lower() for char in word if
char.isalnum())

What kind of syntax is this?

It's a combination of two Python features:

A generator expression:

(char.lower() for char in word if char.isalnum())

This means: "loop through word, keep only alphanumeric chars,
lowercase each kept char."

str.join(...):

''.join(...)
```

This means: "take all those pieces and glue them into one string, with
'' (empty separator) between them."

So together: generate cleaned characters, then join them into cleaned_word.

Read it from inside out

A good way to parse it is right-to-left:

- for char in word: Iterate one character at a time through the input string.
- if char.isalnum(): Keep only letters and digits. Drop spaces, punctuation, symbols (' ', ',', '!', etc.).
- char.lower(): Convert remaining character to lowercase ('A' -> 'a').
- ''.join(...): Concatenate all produced lowercase characters into one string.

Equivalent long-form version

This is the same logic written in a slower, explicit style:

```
pieces = []
for char in word:
    if char.isalnum():
        pieces.append(char.lower())
cleaned_word = ''.join(pieces)
```

If this version feels clearer, that's great — the one-liner is just
a compact form of this.

Concrete example

Suppose:

word = "A man, a plan!"

Character-by-character:

- 'A' -> alnum yes -> 'a'
- ' ' -> alnum no -> skipped
- 'm' -> yes -> 'm'
- 'a' -> yes -> 'a'
- 'n' -> yes -> 'n'

```
- ',' -> no -> skipped
- ' ' -> no -> skipped
- 'a' -> yes -> 'a'
- ' ' -> no -> skipped
- 'p' 'l' 'a' 'n' -> yes -> kept
- '!' -> no -> skipped

Joined result:

cleaned_word == "amanaplan"

That's why I called it "normalization/cleaning": it transforms
many
input formats into one comparable form.

If you want, I can rewrite is_a_palindrome in an intentionally
verbose
beginner-friendly style right in myprog.py (same behavior, easier
to
read).
```

Summary

This chapter covered a lot of ground, from the history of AI to the invention of large-language models, to the practical use of Copilot inside an IDE. You learned how Copilot works, and how to make use of it inside your editor — for code completion, fixing bugs, improving code, and explaining things when you don't understand them. Along the way, you saw why Copilot is such a popular tool among programmers.

If you continue to use Copilot, you'll soon discover that it's able to do more than just complete a few lines of code in a function, or review the code you've written in a file. In Chapter 15, you'll start to experiment with *agentic coding*, an increasingly popular (and controversial) technique for building software using LLMs.

Chapter 15. Agentic Coding

As you learned in Chapter 14, an LLM's training inputs determine what it can produce as output. LLMs can write English sentences because they were trained on English sentences. They're able to write poetry because they were trained on poetry. And they're able to write Python code because they were trained on Python code. The more training a model gets in a particular style, vocabulary, and domain, the more it's able to generate high-quality output in that domain, using that particular style and vocabulary.

This raises important questions: can we ask AI to write software for us? If so, will the quality be any good? Can AI systems replace human programmers? If so, then what is the role of the human in writing software?

In this chapter, you'll learn about *agentic coding*, in which people don't write code themselves, but ask AI to write it for them. You'll learn about the history of agentic coding, some of the thinking in favor of and against it, and then how to get started using some of the most popular agentic coding systems. By the end of this chapter, you will have written some Python code via an AI agent.

Agentic coding represents one of the biggest, fastest-moving, most consequential, and most controversial topics the software world has ever known. Things will continue to change and evolve over the coming years, and these changes will have a profound effect on how software is designed and written, how engineers are hired, and even what sort of training programmers will need. Software engineering always required that you be ready to learn new skills — but agentic coding is requiring that you acquire new mindsets along with those skills. In this chapter, you'll start on your way, and then decide how, and how much, you'll want to embrace agentic coding in your work.

The History of Agentic Coding

As soon as ChatGPT was released, programmers discovered that they could give it instructions:

```
Write a Python function, mysum, that takes a list of integers,
and
returns their sum.
```

Sure enough, the preceding prompt produces a working `mysum` function. Even then, in LLMs' early days, you could ask it to write simple programs, and it would often succeed, or produce a good first draft.

This soon led to a common prediction, namely that "the newest programming language is English." The thought was that you would no longer need to learn to program using a programming language such as Python. Instead, you would be able to tell an AI system what you wanted to do, and it would produce working code.

There were at least three problems with this approach:

- LLMs were indeed able to write code. But the code contained bugs, including references to modules, functions, and methods that didn't exist. And it would often be unnecessarily complex or inefficient, ignoring common Python idioms.
- Remember that computers do what you tell them to do, not what you want them to do? Programmers make mistakes all of the time, and they've been trained to think and write in the precise, clear way that computers need, using programming languages designed for the task. Asking people without this training to write code using a human (and thus imprecise, ambiguous) language is asking for trouble.
- In many ways, the easiest part of developing software is the initial coding. The hard part is maintaining the software over time — fixing bugs, improving performance, and adding new features. So even if AI can write software, there were concerns over whether it

can write readable, maintainable code, let alone work with existing code.

So while it was nice to talk about having LLMs write software based on English-language descriptions written by nonexperts, this seemed unlikely to pan out.

And yet, people kept discussing this as a possibility, and even acting on it: new IDEs such as Cursor and Windsurf cropped up, encouraging coders to let AI write code for them. Models kept writing better and better code, especially when guided by experienced developers.

Of course, many people asked AI to write code for them, even when experts knew — and often said — that this was a bad idea. GenAI companies encouraged such usage, and enhanced the chatbots such that they could create downloadable files containing source code. You no longer had to copy a chatbot's output into an empty editor window; you could just download a Python file. Or a combination of files, in the case of a larger application.

As time went on, AI models continued to improve in all areas, including in coding. Their output still wasn't as good as the best programmers, but it was better than many beginners.

Enough people — coders and noncoders alike — started to ask GenAI to write code for them that it even got a name, *vibe coding*, coined in February 2025 by computer scientist Andrej Karpathy. Lots of people started to vibe code, and were delighted by what they were able to do. However, they often reported that the end product was buggy, insecure, unmaintainable, or some combination of all three. For example, one of the hosts of the *New York Times* "Hard Fork" podcast vibe-coded a mobile hot-tub-maintenance reminder app for his cohost. They were both delighted, until it completely failed several weeks later.

Perhaps vibe coding wasn't going to work for the masses. But it did start to take off among programmers, who found that given appropriate prompts and constraints, AI could actually write decent code. It got a particular

boost when Anthropic released Claude Code, a command-line tool that runs on your computer, rather than on an AI server in the cloud. Because it runs locally, Claude Code can read from and write to files on your computer. This means that you can ask it to write an application, let it churn for a while, and come back to a finished app. Claude Code can also run inside an IDE, going beyond the analysis, advice, and refactoring that you saw in Chapter 14.

The popularity and rapid adoption of Claude Code didn't go unnoticed by Anthropic's competitors. OpenAI soon released their own product, Codex, while Google released one of their own, called Antigravity. All three of these are paid products, with usage limited by the user's subscription level. An open-source alternative, OpenCode, allows you to choose from among a number of free and for-pay model options.

Things Have Changed

By the start of 2026, agent-based coding was really starting to take off: new models released from Anthropic and OpenAI toward the end of 2025 wrote better code than ever before, to such a degree that some commentators, such as Simon Willison and Gergely Orosz, said that soon, writing code by hand would seem as old-fashioned as using punch cards. Wes McKinney, the original author of the pandas library for Python, has spoken about the amount of software he has produced in the Go language — even though he doesn't really know Go, and hasn't written any code by himself.

As I write this in the spring of 2026, the trend seems clear: AI-generated code, increasingly known as *agent-based coding* or *agentic coding*, is not the future, but the present of software development. If you're an experienced programmer, it can allow you to write code faster and more easily than ever before. You can experiment with new ideas and techniques.

In many ways, this represents the next level of abstraction in software development: for years, you haven't needed to think about bits and CPU registers, but have rather thought in terms of high-level data structures and functions. You would code in a programming language, and a compiler

would figure out how to turn that into the appropriate low-level instructions for your operating system and CPU. Agentic coding isn't quite the same, but there is a similarity to it, in that you're now abstracting away the programming language, allowing the agent to decide what code to write. Perhaps, in some ways, the newest programming language is indeed English, as was predicted several years ago. (Actually, AI systems can handle just about any human language. So the newest programming language is whatever you're comfortable speaking and writing.)

Except that there's more to software engineering than just coding, or knowing the syntax of a programming language. Engineering is all about tradeoffs, deciding which features are more important than others and thus deserve more attention. An AI agent can write the code, but it cannot make these sorts of decisions for you. Moreover, AI writes code based on what you have told it, which means that using it effectively means writing clear, precise specifications — including guardrails and hints to ensure that the code is maintainable. An imprecise specification, or a lack of guardrails, is the software equivalent of using an unlicensed contractor to build an addition onto your house. It might work…but it also might not, and there's no telling if and when things will fall apart.

Agentic coding means that programmers are now managers. Your job, in an agentic environment, is to draw up clearly written specifications, to ensure that agents have the skills needed to do their jobs, and then to check on the results. The best software-team managers are able to specify, direct, and check their team's work without necessarily writing the code themselves, and that's how agentic coding works. You might need to dive into the code on occasion, either out of curiosity or for basic quality control. But the real challenge of agentic coding is setting up an environment that ensures the highest possible software quality.

The reason that people are calling this "agentic" coding, and not merely AI coding, is that these programs can run multiple tasks at the same time, each handled by a different AI "agent." One possible use of multiple agents is to implement several new features at the same time. Each agent can implement a separate feature, much as each person on a programming team

can work on a separate feature. Another approach to using multiple agents is to assign them different roles, with one implementing the code, another checking the style and readability, and a third writing and executing tests. In companies that collaborate via GitHub, agents are increasingly writing, checking, and approving patches to existing software, asking for human input only when needed.

The bottom line is that agentic coding — using AI-powered systems, running locally on your computer — to write code is not only increasingly acceptable, but increasingly *expected* by many parts of the computer industry. There are even reports, as of this writing, of companies that have public leaderboards of top employees' AI usage, to encourage everyone to use it more. There are also reports of prospective employees negotiating not only over salary, vacation time, and stock options, but also agentic coding budgets.

Agentic coding is still new, and the best practices for writing software using AI agents are still being worked out. Moreover, it seems increasingly clear that using agents is a new, separate skill. It's not quite the same as programming by hand, although success does depend on having programming experience. It's not the same as managing a team of workers, although thinking about how to divide up a problem does require some of the same thinking.

Objections to Agentic Coding

Agentic coding is becoming more mainstream and popular, but that doesn't mean everyone is happy with it. It's hard to say just how many programmers are resisting agent-based coding, but any mention of Claude Code or the like on social networks almost inevitably provokes negative reactions. Even if you disagree, it's important to know what people's objections are — if nothing else, because there is some truth (and often a lot of truth) in each of them.

And of course, you might find some of these arguments compelling, and decide that agentic coding isn't for you.

Code Quality

One of the main objections to agentic coding is that GenAI-produced code isn't as good as human-written code. This criticism comes in a number of flavors, ranging from the fact that LLMs hallucinate, referencing data structures, functions, and modules that don't exist, to their less-than-fluent, less-than-elegant solutions to many problems. In other words, you shouldn't trust AI to write code, because it might well contain errors.

There is some truth to this, although the models released at the end of 2025 were, by all accounts, a significant improvement over their predecessors, and produced better code. That's why it's only as of early 2026 that so many experts and commentators have taken agentic coding seriously. I myself didn't believe that you could or should trust AI to write code for a long time, for both pedagogical reasons and also for practical ones. The practical reasons seem increasingly quaint, however, in light of improved model quality.

That said, getting high-quality code out of AI isn't just a matter of using a high-quality model. It also depends on well-written specifications. Even then, models will make mistakes, and plenty of them. This is especially true given that LLMs are non-deterministic; even if you give precisely the same instructions on multiple occasions, you'll get different results. That's not inherently bad, given that human programmers will also give you different results each time you ask them to solve a problem. But the non-determinism, along with the ambiguity of specifications and a model's tendency to hallucinate, means that checking your code carefully is of the utmost importance.

How can you check your code? Believe it or not, by writing more code! Automated software testing has been around for many years. The basic idea is that a program invokes your function with a large number of different inputs, knowing what output you should get each time. The automated test then compares the actual outputs with the expected ones. If there are any mismatches, then you know to fix the code.

The most obvious way to test code is to write your functions, then write some tests. If the tests pass, then hooray!

The best technique for testing is known as TDD, short for *test-driven development*:

- Decide what feature you want to implement, or what bug you want to fix.
- Write one or more tests that assume the feature has been implemented or the bug fixed.
- Run the tests. They will fail — because the code hasn't yet been implemented. In many systems, you'll see red icons indicating failure.
- Implement the code.
- Rerun the tests. They should now succeed. In many systems, you'll see green icons indicating success.

This red-green cycle of TDD ensures that everything you have implemented has been tested. And it does tend to produce higher-quality code than when you implement tests after writing the code. However, most people don't have the discipline or patience to write code using TDD.

AI, however, is patient and obedient. It's thus common for people engaging in agent-based coding to tell the AI system that it must use TDD when developing the code. This doesn't guarantee that the software will be without bugs, but it dramatically reduces the potential for them. It also ensures that the AI agents will avoid making many large changes at any point in time, because doing so would make TDD difficult or impossible.

Maintenance

A related criticism is something you've already seen, that it's easier to create a new program than to maintain, modify, or update an existing one. How well will GenAI tools be able to fix software they have written? How well

will people be able to understand the software written by GenAI, if and when human supervision will be needed?

It's certainly possible for AI systems to create unreadable code. After all, they were trained on all of the code on the internet, and much of it is bad. Plus, AI agents don't really know what they're doing, which means that they can produce code which works, but which no person would ever want to read or maintain.

But consider a typical team at a software company: not everyone has the same level of experience, and everyone has a different style of coding. Most companies will thus dictate coding standards and styles, to ensure greater code uniformity. This results in fewer sections of brilliant-but-unreadable code which are difficult, if not impossible, for anyone other than the original author to debug, maintain, and extend.

In the same way, it's possible to instruct AI agents to follow coding standards. You read, in Chapter 6, that Python's PEP 8 gives some style instructions in the Python world. Beyond that, tools such as Black and Ruff have become mainstream in recent years, reorganizing and reformatting code to ensure that it follows best practices.

Sometimes these "best" practices are arbitrary, chosen not because they are right, but simply to ensure that everyone's code is more uniform, and thus easier to read and maintain. For example, should you normally use single quotes or double quotes in your Python code? It doesn't really matter, and PEP 8 just recommends that you are consistent. But Black established the convention of always using double quotes, and many Python programmers now default to them as a result.

So while it's true that GenAI can produce poorly written code, combining test-driven development with standard code formatters tends to reduce the chances of such trouble. Besides, AI doesn't have a fragile ego: you can ask it to review code that it just wrote with a critical eye, and ask it to evaluate its readability. Agentic code review isn't a panacea, but it can help to maintain at least some code quality.

Am I Obsolete?

Some programmers are offended that their work, which they see as a combination of art and engineering, and which they have perfected over many years, can be replaced by a machine. It's a matter of pride, but it's also frustrating to think that a skill that you spent years learning and even more years perfecting might be obsolete.

But consider that a few decades ago, programmers would write in assembly language, without languages like C. They turned their noses up at compiled code, saying that their hand-crafted assembly language was more efficient and maintainable than what a compiler would produce.

But soon enough, two things happened: compilers got significantly smarter and better, and were able to optimize code better than any human could in a fraction of the time. Moreover, chips were designed to be used not by human coders, but by compilers. The result? Today, writing assembly language by hand, when it's even possible, is almost never better than what a compiler could produce.

Did programmers lose their jobs when everyone switched away from assembler, toward high-level languages? Not at all; there has never been more need for programmers. But the type of work that programmers do has changed dramatically, focusing on high-level architecture, data structures, network configurations, and databases rather than counting individual bytes.

Agentic coding does introduce a new set of skills that software engineers will need to learn. Not every job will require it, and not everyone will choose to learn these skills. But it doesn't necessarily mean that programming jobs are going away, or that fewer programmers will be needed. We just don't know yet; the technology is too new, and it's thus too early to judge.

The most pessimistic view of things is that companies will be able to fire most of their software engineers, hiring just a handful of talented people who control thousands of agents that write code. The most optimistic view of things is that companies will expand what they want to do, and hire more

programmers — but that agent-based coding will also enable small groups to create products on their own.

The reality will likely be somewhere between these two. Human developers will still be needed to instruct, manage, and check the output from agent-based coding systems. Plus, on technologies that are less common, less documented, and for which there is less training material, human developers might still do better than AI. However, it also seems likely that using GenAI to write code will continue to be expected, or even mandated, at many companies. It'll no longer seem strange to expect coders (via AI) to produce hundreds, or even thousands, of lines of code in a single workday.

AI Companies Are Problematic

Even if you think that agentic coding is an impressive technological achievement, you might still hesitate at using it when coding, because of the companies behind the LLMs. Among the objections I've heard:

- AI companies didn't get permission from copyright owners when training models.
- AI companies have affiliations with problematic governments and government agencies.
- Information that you submit to a commercial model might be used by the AI company for further training — or simply recorded.
- Many of the companies offering AI are unprofitable; what happens to your development if they go bankrupt?
- The cost of agentic coding adds up pretty quickly, with many engineers spending thousands of dollars each week.

GenAI is not only a new technology, but agentic coding is a new application of that technology. It's not a surprise, given the history of the computer industry, that the leading companies have achieved their success not only through hard work and moving quickly, but also by cutting some

corners. (And in some cases, perhaps even by breaking laws or engaging in bad behavior.)

The good news is that you can engage in agentic coding using open source models running on your own computer. Just as you can download, install, and use Python on any computer and for any purpose thanks to its open source license, you can also download, install, and use open source models. Open source models also remain inside your organization, avoiding any of the potential leakage of secure or proprietary information that might concern you.

The bad news is that open source models are typically less capable than their commercial counterparts. They're likely good enough for day-to-day work, but you'll have to evaluate whether that's indeed the case.

People Won't Learn

One of the biggest questions, and concerns, in the programming world regarding agentic coding has to do with education: senior programmers, coupled with LLMs and agents, are highly effective, thanks to their experience developing software using traditional methods. If everyone uses GenAI, then how and when will recent graduates and junior developers acquire the experience they need?

This is indeed a difficult question, and one for which there aren't any good answers, at least not yet. For now, agentic coding is undoubtedly most effective when done by people who know how to direct the LLM to perform tasks in the most appropriate way. They also know when and how to challenge AI's responses, and even when to interrupt it to better direct a current task.

There are other fields, such as architecture, law, and medicine, where technological tools have dramatically changed the face of the profession over the last century. Tasks that previously took hours, days, or even weeks are now accomplished in a fraction of that time. However, newcomers to the field are still trained in at least some of the traditional methods, so that

they can have a better appreciation for what the automation is doing. It'll take some time before educators in the programming space decide on how much GenAI should be taught, and when.

In some ways, this mirrors a debate that took place in university computer-science programs: some instructors insisted that if you're learning to program, you should start with a low-level programming language like C that forces you to track memory by yourself, reflecting the hardware on the computer. Other instructors said that developing software requires a distinct mindset from working with hardware, and that higher-level languages like Python are more appropriate. The latter group largely won the debate, with an enormous number of universities teaching Python as their introductory programming language. The question of whether, and how, to add GenAI into the curriculum will likely spark similar debates.

Just as this book was going to press, MIT released a report describing how they expect to change the way that they teach and grade students in the era of AI. I am quite impressed by this document, which I believe will have a profound influence on higher education over the coming years.

Getting Started

It's time for you to actually get started with some agentic coding. In this section, you'll install Claude Code, OpenCode, or both. Claude Code is the most popular option among developers, but it does require a paid Claude account. OpenCode is free, open source, and more flexible, and comes with free models with which you can get started immediately. Either will be more than good enough for initial exploration.

Installing and Using Claude Code

Full instructions for installing Claude Code are on the Claude site. You'll typically install it by executing a command from within a terminal window on your computer. The program is installed into your own personal

directory; no administrative permissions are necessary for you to install or run Claude Code.

Once it is installed, you will likely need to restart your terminal program. You'll then need to decide in which directory you'll want to use Claude Code. Use the `cd` command to switch to a directory. For example, on my Mac, I can switch to my *Desktop* folder with:

```
cd /Users/reuven/Desktop/
```

I can then create a new directory with the `mkdir` command, and switch into it with `cd`:

```
mkdir agentic
cd agentic
```

Finally, you can then start up Claude Code with the `claude` command:

```
claude
```

You can then type anything you want to Claude Code. It's the same as when you write to the GenAI system via the Web or on your phone, but with one big difference: it can read and write files on your computer.

> **NOTE**
>
> Most agentic coding systems, including Claude Code, are quite cautious about what directories and files they access, and what programs they run on your behalf. Expect to be asked repeatedly about accessing various diretories and files. It's annoying, but it's far better that they come with such defaults than read and write information from your computer without permission.

You can thus say:

```
Write a simple "Hello, world" program in Python. Store it in a
file
called hello.py.
```

Claude Code will think for a few moments, and will then produce the file, which you can run from outside Claude Code. To exit, just type

```
/quit
```

All commands in Claude Code begin with the `/` character. You can get full help documentation by typing `/help`. But you can also ask Claude Code questions using English — after all, this is the same system you use via a browser or on your phone.

Two more important commands, beyond `/quit` and `/help`, are `/model` and `/usage`. As a paid customer, you are entitled to use a certain number of *tokens* with a Claude model. You can think of a token as a five-letter word. Everything that you submit to Claude is measured in tokens, as is everything that it sends back to you. As of this writing, Anthropic's smartest model is Opus 4.6. However, you can almost certainly get away with another model, Sonnet 4.6, which uses tokens at 20 percent of the rate of Opus. In other words, you can converse with Sonnet five times as long as Opus before you reach your token-usage ceiling. The `/model` command lets you see which model you're currently using, as well as switch to a different one.

You can also check your current usage with the `/usage` command. It'll show you usage for the current model in your current session. It'll also show you how much of your weekly token budget you have used. And finally, it'll show the weekly token budget for this particular model you have used.

Because Claude Code is running on your computer, it potentially has access to all of your files. Claude Code has been designed to ask for permission before reading files from a given directory, writing files in a directory, or retrieving pages from the internet. The first time you ask Claude Code to perform a task for which it doesn't have permission, it'll ask for your approval. You can give one-time approval, or you can say that this particular action is fine both now and in the future.

If you want to cancel something in Claude Code, you can typically press the Escape key.

Installing and Using OpenCode

In many ways, you can think of OpenCode as an open source clone of Claude Code. That's not precisely true, but much of it operates in the same way. You can get download instructions from the OpenCode home page.

To work in a new directory under your Desktop folder, first use `cd` to move into your Desktop:

```
cd /Users/reuven/Desktop/
```

Then (if you didn't do so before), create and enter an *agentic* directory:

```
mkdir agentic
cd agentic
```

Finally, start up OpenCode with the `opencode` command:

```
opencode
```

You can type any English-language command or query you want into the text field in the middle of the screen. By default, OpenCode uses the BigPickle model, which is free and good enough for many small tasks — although it is neither as smart nor as fast as Claude's models. For example, you can give it the following instruction:

```
Write a simple Python program that asks for the user's name and
then greets them.
Store it in greet.py.
```

As with Claude Code, you can quit with the `/quit` command, and get help with the `/help` command.

Unlike Claude Code, OpenCode lets you choose from a wide variety of models, from a large number of vendors. The `/models` command lets you choose models, marking those that are free of charge. If you like, you can select a paid model, choosing from dozens of different options — including those from Anthropic, OpenAI, and Google. Whereas Claude Code only

works with Anthropic's models, OpenCode works with all of them. But there's a catch: on a per-token basis, it's far cheaper to use Anthropic's models via the membership payment than via an API payment using OpenCode. If you're a light user, then you might find that the API payment is less expensive. It's worth researching this a bit before committing.

Exercise: Address Book

Now that you have access to an agentic coding system, it's time for a different kind of exercise, one in which you'll ask GenAI to write software on your behalf. You'll need to specify it clearly, but you won't write the code yourself. Instead, you'll look at the code and see if you can find any issues with it. You'll also ask GenAI to evaluate the code quality.

Specifications

Write a simple address-book program in Python:

- The address book will be stored as a list of dictionaries, with each dict representing one person.
- Each dict will contain four key-value pairs: `first_name`, `last_name`, `email`, and `phone`.
- When the program runs, the user is asked to enter a one-letter command:
  - `a` adds a new person. The user is asked for all four data fields. The entered information is stored in the address-book data structure.
  - `l` lists all people in the address book.
  - `f` finds a person in the address book.
  - `w` writes all of the address-book data to a file on disk.

  - `r` reads data from a file on disk into the program.
  - `q` quits from the program.
- Any command other than those specified results in an error message.

Solution

I entered the preceding text into both Claude Code and OpenCode. I got the following from Claude Code:

```
import json


def add_person(address_book):
    first_name = input("First name: ").strip()
    last_name = input("Last name: ").strip()
    email = input("Email: ").strip()
    phone = input("Phone: ").strip()
    address_book.append({
        "first_name": first_name,
        "last_name": last_name,
        "email": email,
        "phone": phone,
    })
    print(f"Added {first_name} {last_name}.")


def list_people(address_book):
    if not address_book:
        print("Address book is empty.")
        return
    for i, person in enumerate(address_book, 1):
        print(f"{i}. {person['first_name']} {person['last_name']}
"
              f"| {person['email']} | {person['phone']}")


def find_person(address_book):
    query = input("Search by name (first or last):
").strip().lower()
    results = [
        p for p in address_book
```

```
        if query in p["first_name"].lower() or query in
p["last_name"].lower()
    ]
    if not results:
        print("No matching entries found.")
    else:
        for person in results:
            print(f"{person['first_name']} {person['last_name']}
"
                  f"| {person['email']} | {person['phone']}")


def write_to_file(address_book):
    filename = input("Filename to write: ").strip()
    with open(filename, "w") as f:
        json.dump(address_book, f, indent=2)
    print(f"Wrote {len(address_book)} record(s) to {filename}.")


def read_from_file(address_book):
    filename = input("Filename to read: ").strip()
    try:
        with open(filename) as f:
            data = json.load(f)
        address_book.clear()
        address_book.extend(data)
        print(f"Loaded {len(address_book)} record(s) from
{filename}.")
    except FileNotFoundError:
        print(f"File not found: {filename}")
    except json.JSONDecodeError:
        print(f"Could not parse {filename} as JSON.")


def main():
    address_book = []
    commands = {"a": add_person, "l": list_people, "f":
find_person,
                "w": write_to_file, "r": read_from_file}

    while True:
        command = input(
            "\nCommand (a=add, l=list, f=find, w=write, r=read,
q=quit): "
        ).strip().lower()
        if command == "q":
            print("Goodbye.")
            break
```

```
        elif command in commands:
            commands[command](address_book)
        else:
            print(
                f"Unknown command: '{command}'. "
                "Please enter a, l, f, w, r, or q."
            )


if __name__ == "__main__":
    main()
```

I got the following from OpenCode, using the BigPickle model:

```
import json


def add_person(address_book: list[dict]) -> list[dict]:
    person = {
        "first_name": input("First name: ").strip(),
        "last_name": input("Last name: ").strip(),
        "email": input("Email: ").strip(),
        "phone": input("Phone: ").strip(),
    }
    address_book.append(person)
    return address_book


def list_people(address_book: list[dict]) -> None:
    if not address_book:
        print("The address book is empty.")
        return
    for i, person in enumerate(address_book, 1):
        print(
            f"{i}. {person['first_name']} {person['last_name']} "
            f"| {person['email']} | {person['phone']}"
        )


def find_person(address_book: list[dict]) -> None:
    query = input("Enter search term: ").strip().lower()
    results = [
        p
        for p in address_book
        if query in p["first_name"].lower()
        or query in p["last_name"].lower()
```

```
        or query in p["email"].lower()
        or query in p["phone"].lower()
    ]
    if not results:
        print("No matches found.")
    else:
        for p in results:
            print(
                f"{p['first_name']} {p['last_name']} "
                f"| {p['email']} | {p['phone']}"
            )


def write_file(address_book: list[dict]) -> None:
    filename = input("Enter filename: ").strip()
    with open(filename, "w") as f:
        json.dump(address_book, f, indent=2)
    print(f"Saved {len(address_book)} entries to {filename}.")


def read_file(address_book: list[dict]) -> list[dict]:
    filename = input("Enter filename: ").strip()
    with open(filename, "r") as f:
        address_book = json.load(f)
    print(f"Loaded {len(address_book)} entries from {filename}.")
    return address_book


def main() -> None:
    address_book: list[dict] = []

    commands = {
        "a": lambda: "add",
        "l": lambda: list_people(address_book) or None,
        "f": lambda: find_person(address_book),
        "w": lambda: write_file(address_book),
        "r": lambda: "read",
    }

    while True:
        cmd = input("\nCommand (a/l/f/w/r/q): ").strip().lower()

        if cmd == "q":
            break
        elif cmd == "a":
            address_book = add_person(address_book)
            print("Person added.")
        elif cmd == "l":
```

```
            list_people(address_book)
        elif cmd == "f":
            find_person(address_book)
        elif cmd == "w":
            write_file(address_book)
        elif cmd == "r":
            address_book = read_file(address_book)
        else:
            print("Unknown command.")


if __name__ == "__main__":
    main()
```

You can see that these two models both produced working code. And there are certainly some similarities between what they did, as you might expect. But there are also differences; OpenCode's BigPickle model used more advanced techniques, including type hints and `lambda` (something not covered in this book). Both programs read and write their data using JSON.

In both Claude Code and OpenCode, I then wrote:

```
As an experienced, tough software engineer, perform a code review
on the address book program.
```

In each, I got a (long!) list of criticisms ranging from security issues to problems with phone-number formatting. In both cases, I could then ask the agentic system to modify the file, fixing its criticisms. And this is without employing TDD, Black, Ruff, or any other code-testing or formatting program.

Summary

Agentic coding is changing the computer industry, faster and in more ways than anyone can predict. You cannot ignore GenAI code-writing tools, but you must remember that at the end of the day, you're still responsible for the code that is written on your watch. Having a good sense of how to write code by yourself, and not only via AI agents, is a good way to prepare yourself for asking LLMs to write code for you.

In Chapter 16, you'll learn where to go after finishing this book — what you should learn, how you can learn most effectively, and what resources are available to members of the Python community.

Chapter 16. What's Next?

This book has taken you on a whirlwind tour of programming:

- You started with some simple ideas and programs.
- You then learned about built-in data structures.
- You learned how to combine data structures.
- You wrote functions.
- You used modules from Python's standard library.
- You downloaded, installed, and used third-party packages from PyPI.
- You got advice from AI on how to improve your code.
- You used agentic coding to write code on your behalf.

Veteran programmers often like to boast about how hard it was when they started in the computer industry, and how easy newcomers have it. And to a large degree, that's true: when I started programming, most of the topics in this book were significantly more difficult to accomplish. Some of them, such as downloading third-party libraries, were rare and paled in comparison with the depth and breadth of today's PyPI. And getting programming advice from AI agents, let alone asking AI to write code on your behalf? That was the stuff of science fiction.

So yes, it has never been easier to get into programming. Open source software such as Python has made it affordable for anyone with time and motivation to learn to program, and even to distribute code to others. Tools such as PyCharm and VS Code, available free of charge, have made it easier than ever to manage complex software projects. And AI ensures that you can always get feedback on your code, and even assistance in writing it.

But the list of topics you've learned in this book just scratches the surface of what programmers need to know. Over time, you'll likely need to learn many more subjects, from databases to APIs, from object-oriented programming to data analysis. The tech world has always moved quickly, and has always expected that you keep learning. It does feel like the speed of development and change is accelerating, though. This means that while programming tools are better and cheaper than ever, the number of things you're expected to know has grown dramatically. Even to industry veterans, the pace of change can feel overwhelming.

Consider a modern doctor, as compared with a doctor in the 1800s. Two centuries ago, a doctor probably knew everything there was to know about medicine, and could treat a wide set of problems. Today, medicine is infinitely more sophisticated, with diagnoses, equipment, and medicines for problems that they didn't even know existed two centuries ago. The trade-off is that just to graduate from a modern medical school, you need to learn more than the most sophisticated, experienced doctors knew back then. There's too much for any one person to know, which is why doctors now specialize (or even sub-specialize) in a particular type of medical problem. Even after limiting their practice to a particular domain, doctors are constantly working to keep up to date with the latest discoveries and inventions.

This is roughly the state of affairs in the programming world, too: you'll have to choose a specialty, since there's no way for any one person to know everything, or even close to it. You'll need to keep learning during your entire programming career, both so that you can take advantage of new technologies and techniques, and also so that prospective employers will take you more seriously.

The fact that there's so much to learn, and that you'll keep needing to learn throughout your time in the coding world, shouldn't stop you from taking your first steps. You've already seen that even with a small amount of knowledge and experience, you can write programs, and you can get them to work. Sometimes it takes a bit longer than usual, and you'll often make mistakes and have bugs, but that's the nature of coding.

Remember that learning to program is like learning a foreign language: the only way to get better is to (a) use it and (b) make mistakes as you do. There's no such thing as having learned a language completely; there are always new insights and idioms, as well as grammatical oddities, to learn. I've been studying Chinese for a number of years, and I still speak haltingly, with a strong accent and poor grammar. But I can converse with locals in Chinese-speaking countries, to my never-ending delight. Communication, not perfection, is the goal. The same is true with coding: if you can get something done, then you're doing it right. You'll slowly write more efficient, idiomatic, and maintainable code. But for now? The goal is to get things to work, learning the "right" techniques along the way. As you gain experience, you'll gain the confidence you need to create larger, more complex projects.

This chapter is all about how you might want to proceed after finishing this book. How can you learn more, and better? How can you practice? What pitfalls should you watch out for? And what are some topics that you might want to learn next?

By the end of the chapter, you'll have a roadmap for moving forward as a coder, whether it's for work or a hobby. And you'll hopefully realize that regardless of whether you're coding for work or pleasure, coding is a journey, one that involves a great deal of learning and frustration — but also satisfaction and practicality.

What to Learn

Programmers know that there is always something new to learn. Whether it's new ways to write functions, techniques for getting your code to run faster or use less memory, new coding philosophies such as object-oriented programming, new libraries, or even new tools, being a programmer means that you're always learning. If you enjoy learning, then programming will be right up your alley.

But there are numerous topics that you could choose to learn. What *should* you learn next? Where should you focus your limited time and attention, so

as to best improve your skills? The answer, of course, depends in many ways on what you want to do. However, here are some ideas for next steps.

Solidify Your Basic Skills

This book introduced some of the most foundational ideas in programming, including data structures and functions. It's impossible to exaggerate the importance of knowing when and how to use Python's core data structures, including their methods. The best Python coders are intimately familiar with strings, lists, tuples, and dicts, even (or especially) when developing large projects.

After working with these data structures for some time, you'll instinctively know whether to use a list or dictionary, or whether you need to combine them, and how to do so. Python's core data structures have been undergoing continuous improvement for more than three decades, which means that they're both flexible and efficient. If you want to do something with a list or dict, the odds are extremely high that there's a way to do what you want, simply because others probably encountered similar problems in the past.

Choosing the right data structure, or combination of data structures, can have a profound effect on the programs that you write. Taking the time to think about what kinds of data you're storing, how you'll add or modify that data, and how you plan to retrieve it, are all important to ask *before* you start coding.

Once you feel comfortable with data structures, make sure that you also feel fluent writing functions. Solving problems often means breaking them into smaller, more manageable problems, and functions are the main way to do that in software. Designing a flexible function that is easy for users to understand and also for maintainers to improve is challenging, and takes practice. You learned some of the most common ways in which arguments are assigned to function parameters; Python offers several more variations on this theme, and it's worth learning about them to build truly interesting, useful functions.

Object-Oriented Programming

When you work on a large project, it's easy to end up with hard-to-follow, complex combinations of data structures, and an even larger number of functions that manipulate them. It might work, and might even be efficient, but maintaining that kind of program can be particularly difficult.

In the early 1980s, programmers started to embrace a new way of writing code known as *object-oriented programming*. The idea was that by writing new, higher-level data structures, your code would be easier to write, easier to think about, and easier to maintain. Each of those data structures would then have its own methods, giving you a package of nouns and verbs that were guaranteed to work together.

Object-oriented programming is now a mainstream way of approaching software design. Some programming languages, such as Java and C#, even require that you write your code in an object-oriented style. Python is implemented using objects, but it doesn't mandate that you write your code using those techniques.

It's probably a good idea for every Python developer to learn object-oriented programming at some point. It's a useful technique to have in your arsenal. Plus, so much of Python, including such built-in types as strings and dicts, is built using objects that you'll get many insights into how Python works.

But do you *need* to learn objects to succeed as a Python programmer? Not really:

- For mainstream applications, you'll almost certainly want to use objects, if only for convenience.
- For web applications using the Django framework, you'll also need objects. Other frameworks, such as Flask and FastAPI, make it less necessary, but still convenient.
- For system and network administration, you're unlikely to need objects. However, some of the packages you use from PyPI will be

written in an object-oriented style.

- For data science, you won't need objects very much. You'll be using objects written by other people, but that won't be much different from using strings, lists, and dicts.

It's important to understand that object-oriented programming doesn't change what software can do. It's a code-management technique, one that comes with a slew of new vocabulary words and concepts. But at the end of the day, it's all about data structures and functions, and using them to make problems easier to understand, break apart, and solve — and certain problems need objects more than others.

Data Analysis

The world is awash in data. Whether it's weather records, economic reports, or the performance of your ecommerce business, there's a lot of data that organizations need to process. They need people to perform that analysis, and those people need software to help them do their jobs. Increasingly, that tool is Python.

Indeed, for data professionals from data analysts to machine-learning specialists, Python is increasingly their go-to language. This includes people who have long used tools like Excel, SQL databases, and even languages like Matlab.

It might seem strange to use Python, given that it's not a particularly efficient language. Python uses more memory than many other systems, and takes longer to execute. How, then, is Python the most popular language among data analysts, data scientists, data engineers, and machine-learning specialists?

The secret is NumPy, a package that is mostly written in C, but with a thin Python layer that lets you use it from within your Python programs. A large number of data-analysis libraries have been built on top of NumPy, including pandas, a full-fledged data-analysis library that lets you read, clean, analyze, export, and visualize data with relative ease.

Using Python for data analysis means using the special-purpose data structures defined by NumPy and pandas, instead of the more familiar lists, tuples, and dicts that you learned about in this book. They can take some time to learn, not because they're hard to use, but because they're different, and require that you think in different ways. Learning when to use the core Python data structures, and when to use the special NumPy and pandas data structures, also requires some patience and practice.

Given Python's popularity in the world of data, it's no surprise that a huge number of Python jobs now expect at least some familiarity with NumPy and pandas. Indeed, some jobs are mostly spent using pandas, reading data from an ever-growing number of sources and then trying to make sense of that data for upper-level management, customers, or both.

While the ideas associated with data analysis can be a bit complex, the code you have to write when working with NumPy and pandas tends to be much simpler and shorter than with garden-variety Python. You write many fewer functions, depending on the many specialized methods that these packages provide. You also barely use `if` and avoid `for` loops, which mess with the speed optimizations (known as *vectorization*) that NumPy provides.

Also, if you're planning to use Python for data analysis, then you likely won't be defining your own classes. It's just not something that comes up nearly as often in the world of data analysis, because you're so dependent on arrays (in NumPy) and data frames (in pandas), which provide most of the functionality you'll want and need. This means that while object-oriented programming is good and useful to learn, it's less important for data analysts and engineers than for many other Python developers.

Automated Testing

You've written a function. Does it work correctly?

You've written a few dozen functions. Do they all work correctly?

You've written a full application. Does it work correctly when faced with real users, who are known for giving weird and illegal inputs?

Long ago, the only way to answer these questions was to manually go through a program, checking each of the functions manually. Then, when the program was complete, one or more quality-assurance testers would then try out the program, following a script and noting where things went wrong.

But today, there's an easier way, using *automated testing*. Instead of you calling a function and checking its results, you write a test program that calls the function. Because computers are so much faster than people, your test can check what happens when you invoke the function with a number of different arguments and inputs.

This is especially useful as a program gets large and complex. You modify the program in one place, and don't realize that it has affected something in a totally different place. Automated testing doesn't mean that a program is free of bugs, but it does reduce them, as well as ensure that no known, previously fixed bugs return.

Python supports a number of different ways to test your programs, but the most popular is known as pytest. The good news is that `pytest` allows you to write your tests using Python functions. The bad news is that `pytest` takes advantage of everything Python has to offer, meaning that your tests won't look like most functions you've seen before. But the strangeness eventually wears off, and allows you to write tests quickly and easily, increasing the robustness of your software.

Most companies now expect their programmers to write not only their code, but the tests that check their code, as well. And if you're using an agentic framework, such as Claude Code? Then it's standard to tell Claude Code to write tests along with the program. (Actually, it's even common to ask Claude to write tests *before* it starts to write the program, in what's known as *test-driven development*.)

Learning about software testing is worth the effort. It really does improve the robustness and stability of your code, and allows you to feel more confident that it's doing what you think it's doing.

Version Control with Git

You've written a program, and something that used to work no longer does. Do you have the previous version of your program, so that you can compare it with the current version, and figure out what you broke?

If you're like many people, the answer is no. Or maybe you keep around older versions of your files with "old," "old2," and similar words and phrases tacked on. If you use such a system, then it's hard to know what work was done when, and whether "old" is older than "previous," and the like.

For decades, programmers have used *version-control systems* to keep track of old versions of their code. Modern version-control systems go beyond this, allowing you to collaborate with other people, even on the same files, without having to worry about who has changed what, or how to resolve conflicts. They let you develop new features, even experimental ones, without affecting the main code that other members of your team are releasing to production.

Version control is thus an important part of modern software development. Today, the most popular version-control system is known as Git. The good news is that Git is extremely flexible, powerful, and fast.

The bad news is that Git was developed by the team that develops the most technically challenging part of the Linux operating system, known as a *kernel*. They're very smart, but they didn't have normal, everyday users in mind when they developed Git — and it shows. Each command can be used for multiple, different things. Multiple terms refer to the same thing. And many of the commands have odd names and seemingly unintuitive options.

Moreover, Git does things very differently from other version-control systems. Companies that adopt Git often tell their employees that it's easy to use Git, so long as you stick with a handful of important, simple commands. People start to use those commands, make mistakes, and get angry.

If so many people get upset with Git, then how has it become so popular? Partly because it's so fast and capable. But partly because a number of

websites have cropped up over the years that allow you to collaborate with your colleagues — or with other developers on the internet — using Git. These include the well-known GitHub, but also GitLab and Bitbucket. Many companies have standardized on GitHub for in-house collaboration, because of the many tools it offers for managing a large software project.

Sooner rather than later, you should spend some time to learn Git. Don't just learn the syntax and commands; spend time learning how Git actually works under the hood. Knowing how it was implemented, how *branches* and *commits* are implemented, and how to both *branch* and *merge*, are skills that will pay off handsomely. You should also learn how to submit and work with *pull requests*, which allows one user to suggest that another user incorporate a change.

But be warned: Git has a famously steep learning curve, and if you find yourself frustrated, know that you aren't alone. Find a Git tutorial or course (or friend) who can explain the underpinnings to you, and everything else will fall into place.

How to Learn

The technology industry has always moved quickly. But with each successive era in technology — the internet, mobile devices, cloud computing, and now AI — the rate of change seems to increase even further. It's hard enough to learn what you need to break into the world of technology. With things changing as fast as they are, how can you even hope to keep up?

Answer: you can't keep up, and don't feel bad that you can't. There's too much to learn, and it's all happening too fast. If you're feeling overwhelmed, rest assured that you aren't alone. A lot of other people also feel overwhelmed, including many of the smart, accomplished people you look up to.

What you *can* do, however, is try to learn as much as you can, as quickly as you can, and to be strategic about that learning. Since you're reading this

book, you're clearly interested in Python. Keep learning more about Python, diving more deeply into the areas that are of particular interest to you.

Here and there, you'll also want to learn about other languages, libraries, and techniques. You won't necessarily need to know those things, but just as learning a foreign language can give you insights into your native language, learning about other technologies can help you appreciate and understand Python better. If you spend 20 percent of your time learning about new, unrelated topics and 80 percent focused on Python, and the areas of Python where you want to improve, you'll soon find yourself understanding things better and able to solve problems more quickly.

Over time, you'll also find that the terminology used in the Python world starts to make more sense. This is particularly important, because the documentation for Python is expansive and well written, but also assumes that you already know a great deal about Python. This becomes something of a catch-22, where you need to read the documentation to understand Python better, but you need to understand Python better to read the documentation. As you gain more experience with Python, you'll slowly absorb the jargon used in Python's documentation and community, and will be able to make use of more resources over time.

It's important to realize that time and consistent practice are your best friends. Don't expect to use Python once every few months and to really absorb its usage patterns. If you spend some time with it each week, though, then you'll find yourself thinking in Python, gradually and over time.

This section offers some ideas for how to improve your Python skills. Any of them is probably fine, and a combination is likely better. Your schedule, interests, and learning patterns, along with a great deal of experimentation, will help you find your path.

Projects

Whenever a famous author is asked how one can improve their writing, the author almost inevitably says that they should write more. This is true; the more you write, the better you get at writing. While people normally say

this about writing in a human language, that happens to be true about programming languages, too.

So yes, you can and should write as much Python as you can. The more you write, the better you'll get. You'll get better at using Python's built-in data structures and functions. You'll also get better at structuring your code for improved readability and maintainability. You'll also learn about tools that help to keep your code readable, including `black` and `ruff`.

What sort of project should you choose? It really doesn't matter. Even the simplest software projects become large and complex. Being forced to structure your code to be more readable and maintainable, and to incorporate tools that encourage best practices is a great thing, and will help you on every project you create after.

The thing is, learning these techniques is difficult if you're working on your own. That's why it's often a good idea not just to work on your own personal projects, but to collaborate with others. The open-source community has many thousands of projects that are looking for help. They often provide a terrific way for you to not only learn, but to do so under experienced coders' guidance. The community gets software, and you get experience — a great deal for everyone!

Of course, not every open source project is created equal. Some are huge and established, and even have guidelines for how newcomers can be incorporated into the group. Others are run by a handful of loosely coupled developers. Still others are one-person operations.

If you're looking to gain experience, then you should probably try to contribute at one of the larger, more established projects. Such projects often have mentors available to help newcomers, as well as bugs that are relatively easy to fix, and are thus good starting points for newcomers.

It's also important to say that you might encounter some jerks in your quest for a useful, interesting open source project. (Many projects have codes of conduct, with penalties for people who abuse others on the team.) The beauty of open source is that you're a volunteer, contributing time and energy because you want to do so, not because you have to. If you aren't

treated well, or if you aren't getting the feedback you need, then it's totally OK to leave the project.

Reverse Socratic Method

You've already learned about, and used, the Reverse Socratic method in this book — where, in order to learn about a topic, you ask GenAI numerous questions. Only after you feel satisfied with your understanding do you stop. Along the way, you ask for examples and clarifications. You also summarize what you think you understand, and then ask AI to tell you whether that summary is correct.

This method can be used on nearly any topic in programming. I personally use it every day, as I'm learning new topics, or even making sure that I understand topics that I haven't taught in a while. Just forcing yourself to formulate intelligent questions requires that you reflect, think, and learn. Being in a long conversation with AI about the topic, asking for it to explain something another time or with a different example, is a great way to ensure that you understand something.

And so, when you inevitably are confused by something, or just want to understand something better, fire up AI and ask it.

By itself, that's not a special technique. People ask billions of questions of GenAI systems every day. The Reverse Socratic part of this is not to stop after one question, but to keep asking. Drill down. Challenge. Ask for examples. Give an example of what you think contradicts what the AI system has told you, and ask it to explain.

The more questions you ask, the more you'll learn. Also, the more you can be sure that the answers you're getting from AI are accurate. It's not at all uncommon for a chatbot to answer a question with a seemingly reasonable answer. But as I ask more and more pointed questions, I discover that actually, what it said isn't true — or isn't as cut-and-dried as it made things sound.

If I get a warning or error message I don't understand, I don't just use AI to help me debug it: I also use AI to help me understand why I got this message, what it means, and how I can avoid it in the future.

I even use this technique on topics that I feel I do know, just to make sure that I'm not missing too many subtle details. Having an infinitely patient (if flawed) source of knowledge that I can pepper with nonstop questions has helped me to improve as a developer and also as an instructor.

For example, let's say that I want to better understand how arguments are assigned to parameters. I prompt Claude:

```
Let's say that I have a really simple function:

def add(a, b):
    return a + b

I then invoke the function:

add(10, 20)

How does Python assign the values 10 and 20 to a and b?
```

Claude answers:

```
Great question — this gets at something fundamental about how
Python
works under the hood.

When you call `add(10, 20)`, Python does roughly the following:

1. **Evaluates the arguments** at the call site. `10` and `20`
are
evaluated to integer objects (in CPython, small integers like
these
are actually cached/interned, but that's a side detail).

2. **Creates a new local namespace** for this invocation of
`add`. Each function call gets its own fresh local scope,
implemented
as a frame object with a local variables dictionary (or, more
efficiently in CPython, an array of "fast locals" indexed by
position).
```

```
3. **Binds the arguments to the parameter names.** Python looks
at the
function's signature — `def add(a, b)` — and matches the
positional
arguments in order: the first argument (`10`) gets bound to `a`,
and
the second (`20`) gets bound to `b`. "Bound" here means the names
`a`
and `b` in the local namespace are made to refer to those integer
objects.

The key conceptual point: Python's argument passing is sometimes
called **"pass by assignment"** (or "pass by object reference").
It's
not pass-by-value (Python doesn't copy the objects) and not
pass-by-reference in the C++ sense (you can't reassign the
caller's
variable through the parameter). Instead, the parameter name
inside
the function is bound to the *same object* the caller passed in.

For immutable objects like integers, this distinction doesn't
matter
much — you can't mutate `10` anyway. But it matters a lot for
mutable
objects: if you pass a list and the function calls `.append()` on
it,
the caller sees the change, because both names point to the same
list
object.

So in your example, after the call begins, `a` and `b` are simply
two
new local names in the function's frame, both pointing to integer
objects that happen to be `10` and `20`. The `return a + b` then
evaluates `a + b` to a new integer (`30`) and returns it to the
caller.

Want me to go a level deeper — into the actual bytecode
(`LOAD_FAST`,
`STORE_FAST`) or the frame object internals?
```

I then respond:

```
Wait, before we go deeper, what if I do this:

x = 10
y = 20
add(x, y)

How is that different? Are x and y, the variables, passed? Does
add
even know that I'm passing x and y?
```

Claude responds:

```
Excellent question — and the answer reveals something important
about
Python's model of variables.

**No, `add` has no idea that `x` and `y` exist.** From `add`'s
  perspective, the two calls are completely indistinguishable:

```python
add(10, 20)
add(x, y)
```

Here's why. In Python, a variable name like `x` is just a label
in
some namespace that points to an object. When you write `x = 10`,
you're not creating a "box" called `x` that contains `10`. You're
creating an integer object `10` and binding the name `x` (in the
current namespace) to refer to it.

When you then write `add(x, y)`, Python does this at the call
site:

1. **Evaluates the expression `x`**, which means "look up the
name `x`
in the current namespace and get the object it refers to."
Result: the
integer `10`.

2. **Evaluates the expression `y`** the same way. Result: the
integer
`20`.

3. **Calls `add` with those two objects** as positional
arguments.
```

By the time `add` starts executing, all it receives are two objects. The names `x` and `y` were resolved and discarded at the call
site. Inside `add`, those objects get bound to the local names `a` and
`b` — and `add` has no way of knowing they were also bound to `x` and
`y` in the caller's scope.

This is why you sometimes hear Python described as having **"names and
objects"** rather than "variables and values." A name is just a reference; multiple names in different (or even the same) namespaces
can point to the same object simultaneously. After `add(x, y)` is called and execution enters the function, the integer `10` has at least two names pointing to it: `x` in the caller's scope, and `a` in
`add`'s local scope. They're peers — neither is the "real" name.

A nice consequence: this is also why reassigning `a` inside the function (e.g., `a = 999`) does *not* affect `x` in the
caller. Reassignment just rebinds the local name `a` to a different
object; it doesn't reach back to the caller's namespace. The caller's
`x` still points to `10`.

If you want to see this concretely, `id()` is illuminating:

```python
x = 10
print(id(x))  # some number, e.g., 4302938384

def add(a, b):
    print(id(a))  # same number!
    return a + b

add(x, 20)
```

The `id()` values match because `x` and `a` refer to the exact same
object in memory.

Ready to go into the bytecode now? That'll show you the `LOAD_FAST` /

```
`LOAD_GLOBAL` instructions that do this name resolution at the
call
site.
```

Notice that I didn't simply say "yes" to Claude, but that I found something that didn't make sense, and followed up with a new question. Remember that you're in the driver's seat, and should be directing the questions to ensure that you understand what's happening. The AI system will always suggest a new question or direction, but it's up to you to decide whether that makes sense. Also: AI isn't a person. You can also ignore its follow-up questions without feeling guilty.

Conferences and Meetups

Python core developer Brett Cannon famously said, "I came for the language, but stayed for the community." His point is that the people who work on, use, and support Python are an even more compelling reason to use the language than the language itself. Python has a famously open, welcoming culture, and that is especially on display at the numerous Python conferences that take place each year. The oldest and biggest of them is PyCon US (formerly just "PyCon"), with EuroPython a decent runner-up. But nearly every country, as well as numerous states, provinces, and regions have their own Python conferences.

Attending a Python conference is a great way to learn. Most obviously, there are numerous tutorials and talks to attend, often by the best-known coders and teachers in the Python community. And the talks are always meant for a mix of levels; you don't have to be a famous Python expert to attend most of them. On the contrary, many of the talks are aimed at beginners, hoping to give them the understanding they need to enjoy and use Python more, and to participate in the community, as well.

Meeting and chatting with people, sometimes known as the "hallway track," is no less important and useful. You never know who you might meet, and what you will learn from a 10-minute chance encounter. As an example, I happened to meet a key developer of the browser-based Jupyter Lite

notebook at EuroPython several years ago. I took the opportunity to describe a problem I was having with it. He explained what the issue was, and how they were trying to solve that issue. Sure enough, a year or so later, I found that the problem had been fixed.

Many Python conferences include *sprints*, in which participants work on a variety of open source projects. Sprints are famously welcoming, and are a great way to get free personal mentoring from some of the best Python developers in the world, and to contribute to large, visible projects that have a real impact.

If you really want to accelerate your learning at a conference, propose a talk. It might sound odd, in a book aimed at first-time programmers, to suggest that you teach others something about Python. But it's well known (if something of a cliche) that the best way to learn something is to teach it. If you have learned something new about Python, or have developed some interesting software, then others would probably benefit from learning about it, too. Just proposing a talk can be a useful learning experience, and many of the larger conferences offer volunteer mentors to help you with your proposal and talk delivery.

There are also thousands of local and regional Python meetups, often getting together every month. You can think of this as a shorter, smaller conference. And if you're a bit put off by presenting at a conference, then meetups are almost always delighted to have new speakers and volunteers.

Traditional Sources

Of course, you can also learn from books, newsletters, and videos. There are numerous books about Python, some (like Al Sweigart's *Automate the Boring Stuff*) that teach via small, practical projects, others (like my own *Python Workout*) aimed at helping you improve via exercises, and still others (like *Fluent Python*, by Luciano Ramalho) that teach more advanced techniques. And of course, many specific techniques (e.g., object-oriented programming) and libraries (e.g., pandas) have their own books.

There are also numerous Python newsletters to which you can subscribe. Python Weekly and PyCoder's Weekly contain links to the latest news, articles, and videos in the Python world. A number of newsletters send one article each week:

- Python tips, from Trey Hunner
- Mostly Python, from Eric Matthes
- The Python Coding Stack, from Stephen Gruppetta
- Mathspp Insider, from Rodrigo Girão Serrão
- My own Better Developers

If you want to hear about the latest goings-on among Python's core developers, and what changes they are planning and making in the language, you might want to read Core Dispatch, from core developer Savannah Ostrowski.

For such a popular programming language, there are surprisingly few podcasts about Python. The biggest and best-known one is Talk Python to Me, with Michael Kennedy. Michael, along with testing expert Brian Okken, also produces a weekly news-about-Python podcast, Python Bytes.

Finally, there are numerous YouTube channels from which you can learn Python. Here are a few good ones:

- ArjanCodes
- Real Python
- Corey Schafer
- My own channel

Beyond the YouTube channels, there's also PyVideo, which collects as many Python-related conference videos as they can. Regardless of the topic or level, you're likely to find numerous useful resources at PyVideo.

Frustration

Programming, when things go well, can be exciting and fun. You get a huge dopamine rush when your program works, doubly so if other people get to use and enjoy your code. It combines engineering and design, with a large dollop of puzzle-solving, all with a practical benefit.

But much of the time, programming can be frustrating. How to solve a problem might not be obvious. More often, you think that you've solved a problem, but it isn't working. Or it's working, but it takes too long to execute. Or it uses too much memory. Or it isn't communicating its results to another part of the program.

Worst of all are bugs that are difficult to replicate. Time zones, especially when countries change their clocks, are notorious for giving programmers headaches. Anything involving input from users is guaranteed to generate strange, unexpected inputs. And sometimes you'll get bad data from a server on which you're relying.

It's important to remember, from the time you start coding, that this kind of frustration is inevitable. It's part of the process, and you might even say it's a *necessary* part of the process. Which doesn't make it fun, not by a long shot.

Years ago, I heard an interview in which an expert programmer said that everyone gets stuck when writing software. The difference between novices and experts is how they react to this situation: new coders think that they're unable to solve the problem, and may be incompetent or in the wrong line of work. Veteran programmers know that they've been stuck before, and that if they just attack the problem in the right way, they'll solve this problem, too.

Indeed, a great icebreaker you can try when attending a Python conference is to ask people about the worst bug they ever encountered, or the biggest mistake that they ever made. You can be sure that anyone with even a few years of experience writing code will have no shortage of such stories. Knowing that you're not the only one making such mistakes is often a relief.

As you get ready to start writing real code, it's important to remember that there will be bumps along the way. Be patient with yourself as you try to puzzle through the problem. If AI can help you to find your bug, that's great — but don't forget to ask it follow-up questions, so that you'll internalize what the problem was, and how it happened. That'll give you experience, and reduce the chances of such trouble in the future.

By the way, you could argue that while AI does offer new ways to alleviate frustration, it also introduces new ones. You have to be very careful that the answers you get are accurate, which often isn't easy when you're not an expert. Even when it writes code, AI can suggest non-idiomatic, slow, or convoluted code. I've certainly found myself using AI to rewrite buggy code, only to discover that things are now worse, not better, than they were before. This is yet another reason why version control (e.g., Git) is an important skill to learn; by committing to version control on a regular basis, you make it possible to return to a previous point in time. That can come in handy if things are messed up by the AI system.

Summary

In this chapter, you learned why it's important to keep learning, and some strategies for doing so throughout your coding career. You also learned that programming is something of a roller coaster, with lots of excitement and positive aspects, but also inevitable frustration. You might think that the more you learn, and the more experience you gain, the less frustration you'll feel. And that's true, so long as you continue building software of the same size and scope. But most of us take our experience and use it to build larger and more complex programs, which leads to more frustration. Know that this frustration is real, but it's also universal among coders. And it doesn't change the fact that programming can be exciting, interesting, and useful — as well as part of a career.

Index

Symbols

A

B

C

D

E

counting with pathlib and Counter, Exercise: Using the Standard Library-Solution

counting words with a dictionary, Exercise: Counting Words-Solution

creating a uv project, Exercise: Using rich-Solution

defining a function with user input, Exercise: Calculator-Solution

exploring a module with dir, Exercise: Exploring a Module-Solution

guessing game with random.choice, Exercise: Guess the Name-Solution

if/elif/else with user input, Exercise: Guess My Age

importing names with aliases, Exercise: Password Generator-AI Challenge: Prompt and Solution

installing and using a PyPI package, Exercise: Installing a Package-Solution

passing arguments to a function, Exercise: Calculator with Arguments-Solution

positional and keyword arguments, Exercise: Fancy Calculator-Solution

powers of 10 with enumerate, Exercise: Expand Numbers-Solution

price lookup with a dictionary, Exercise: Pizza Toppings-AI Challenge: Prompt and Solution

returning a value from a function, Exercise: Calculator with Return Values-Solution

specifying a program for an AI agent, Exercise: Address Book-Solution

summing line lengths in a file, Exercise: File Size-Solution

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

About the Author

Reuven M. Lerner is a full-time Python trainer, in business since 1995. He offers courses, exercises, and mentoring via *LernerPython.com* and teaches for companies such as Apple, Cisco, Intel, and Sandisk. He's also a frequent instructor on the O'Reilly platform, teaching classes such as "Python First Steps" and "Object-Oriented Python Bootcamp."

Reuven is a frequent conference speaker, having presented at PyCon US, EuroPython, PyCon Taiwan, PyCon India, PyCon Israel, and PyData Tel Aviv numerous times. His previous books, *Python Workout* and *Pandas Workout*, were both published by Manning.

Reuven publishes both "Better Developers," a newsletter exploring the Python language, and "Bamboo Weekly," a newsletter with pandas challenges based on current events and real-world data sets.

Reuven has a bachelor's degree in computer science and engineering from MIT and a PhD in learning sciences from Northwestern University. He lives in Modi'in, Israel, with his wife and three children.

Colophon

The animal on the cover of *AI-Assisted Python for Nonprogrammers* is the common eland (*Taurotragus oryx*). Found in southern and east Africa, elands are known for being the largest antelope species in the world.

Common elands are massive. Females weigh between 660 and 1,320 pounds and measure 6 to 9 feet in length; males (also called bulls) can weigh between 880 to 2,200 pounds and measure 7 to 11 feet in length. Despite their great size, they are surprisingly agile and can jump up to eight feet high. Their coats are smooth; females have a tan coat, and males have a darker coat with a grayish tinge. Both have impressive, tightly spiraled horns. Males have thicker horns, which are used during mating season to wrestle other bulls; females use their horns to protect their young from predators.

Common elands avoid dense forests, preferring to live in open plains, savannas, woodlands, and grasslands. They are herbivores and primarily eat grass, leaves, and fruit. They consume water voraciously when it's available, but they can go without during the dry seasons by increasing their body temperature.

Currently, common elands are classified as Least Concern by the IUCN, with 50% of the population living on protected lands. However, they still face many threats, including poaching and habitat destruction due to agricultural expansion.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by José Marzan Jr., based on an antique line engraving from *Natural History of Animals*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.