LEARN

AI-Assisted
PYTHON PROGRAMMING

With Copilot and ChatGPT

Leo Porter • Daniel Zingaro

MEAP

MANNING

LEARN

AI–Assisted
PYTHON PROGRAMMING

With Copilot and ChatGPT

Leo Porter • Daniel Zingaro

MEAP

MANNING

Learn AI-Assisted Python Programming

MEAP Edition

Manning Early Access Program

Learn AI-Assisted Python Programming

With Copilot and ChatGPT

Version 1

Copyright 2023 Manning Publications

https://livebook.manning.com/#!/book/learn-ai-assisted-python-programming/discussion

For more information on this and other Manning titles go to

manning.com

welcome

Thank you for purchasing the MEAP for *Learn AI-Assisted Python Programming*.

We're at the start of a new era in programming. AI coding assistants are here —ChatGPT, Copilot, and others—and they're transforming how people program. The two of us have spent our careers researching and enhancing student learning of programming. We have never been more excited about the possibilities that AI coding assistants bring for the beginner or would-be programmer.

In this book, we fully embrace these tools and use them to offer an alternative way to learn how to write software. We believe that new learners of Python shouldn't be spending time learning low-level details that are now mostly solved by the AI coding assistants. Instead, we believe that these tools empower you to write larger and more powerful software faster.

This book is for those who are interested in learning how to write software with Python or for the many people who tried and failed (we get it!) to learn how to write Python code in the traditional way. No prior programming skills are required, but we expect you to be comfortable with installing software and managing files.

We've organized the book to get you started writing software with Copilot right from the start. Then we teach you the essential skills that you need to program with Copilot effectively: code reading, problem decomposition, and testing. After that, we'll bring all the skills together to write small software projects in meaningful domains.

At the same time, we need to caution you: it's early. These tools seem to be changing on a daily basis. We don't know what programming will look like when the dust settles. The two of us are exploring new territory with few established answers and we are excited to start this journey with you.

If you have any questions, comments, or suggestions, please share them in Manning's [liveBook Discussion forum](#) for our book.

—Leo Porter and Daniel Zingaro

In this book

1 Introducing AI-Assisted Programming with Copilot

This chapter covers

- AI-assisted computer programming
- How AI-assistants change how new programmers learn how to program
- Why programming is never going to be the same
- How AI assistants like Copilot work
- How Copilot can solve many classic introductory programming problems
- What else you can do with an AI coding assistant
- Possible perils of AI assisted programming

In this chapter, we'll talk about how humans communicate with computers. We'll introduce you to your **AI Assistant**, Copilot, an amazing tool that uses Artificial Intelligence (AI) to help people write software. More importantly, we'll show you how Copilot can help you learn how to program. We're not expecting that you've written any programs before. If you have, please don't skip this chapter, even if you already know a little bit about programming. Everyone needs to know why writing programs is different now that we have AI assistants like ChatGPT and Copilot, and that the skills we need to be effective programmers change. As we'll see, we also need to be vigilant, because sometimes tools like ChatGPT and Copilot lie.

1.1 How we talk to computers

Would you be happy if we started by asking you to read and understand this?
[1]

```
section .text
global _start
_start:
    mov ecx, 10
```

```
    mov eax, '0'
    l1:
    mov [num], eax
    mov eax, 4
    mov ebx, 1
    push ecx
    mov ecx, num
    mov edx, 1
    int 0x80
    mov eax, [num]
    inc eax
    pop ecx
    loop l1
    mov eax, 1
    int 0x80
section .bss
    num resb 1
```

That monstrosity prints out the numbers from 0 to 9. It's written using code in assembly language, a low-level programming language. Low-level programming languages, as you can see, are very far from languages that humans can easily read and write. They're designed for computers, not humans.

No one wants to write programs like that. But, especially in the past, it was sometimes necessary. Programmers could use it to do exactly what they wanted the computer to do, down to individual instructions. This level of control was needed in order to squeeze every bit of performance out of underpowered computers. For example, the most speed-critical pieces of 1990s computer games such as Doom and Quake were written in assembly language like the code above. It simply wouldn't have been possible to make those games otherwise.

Making it a little easier

Okay, no more of that. Let's move on. Would you be happier with this?

```
for num in range(0, 9):
    print(num)
```

This, by contrast, is code in the Python language and is what many programmers use these days. Unlike assembly language, which is a low-level

language, Python is considered a high-level language because it's much closer to natural language. Even though you don't know about Python code yet, you might be able to guess what this program is trying to do. The first line looks like it's doing something with the range of numbers from 0 to 9. The second line is printing something. It may not be too hard for you to believe that this program, just like the assembly language monstrosity, is supposed to print the numbers from 0 to 9. Unfortunately, something is wrong with it.

While this code is closer to English, it isn't English. It's a programming language that, like assembly language, has specific rules. As in the code above, misunderstanding the details of those rules can result in a broken program.

The holy grail of communicating with a computer is to do so in a natural language such as English. We've been talking to computers using various programming languages over the past 70 years not because we want to, but because we have to. Computers were simply not powerful enough for the vagaries and idiosyncrasies of a language like English. Our programming languages improved—from symbol soup assembly language to Python, for example—but they are still computer languages, not natural languages.

This is changing.

Making it a lot easier

Using an AI assistant, we can now ask for what we want in English and have the computer code written for us in response. To get a correct Python program that does actually print the numbers from 0 to 9, we can ask our AI assistant (Copilot) in normal English language like this:

```
# Output the numbers from 0 to 9
```

Copilot might respond to this prompt by generating something like this:

```
for i in range(10):
    print(i)
```

Unlike the example we showed you before, this piece of Python code actually works!

AI coding assistants can be used to help people write code. In this book, we will learn how to use Copilot to write code for us. We will ask for what we want in English, and we will get the code back in Python.

More than that, we'll be able to use Copilot as a seamless part of our workflow. Without tools like Copilot, programmers routinely have two windows open: the one where they write code and the other where they ask on Google how to write code. This second window would have Google search results, Python documentation, or forums of programmers talking about how to write code to solve that particular problem. They're often pasting code from these results into their code, then tweaking it slightly for their context, trying alternatives, and so on. This has become a way of life for programmers—but you can imagine the inefficiency here. By some estimates, up to 35% of programmer's time may be spent searching for code [1], and much of the code that is found is not readily usable. This will be much improved with Copilot helping us write our code.

1.2 About the technology

We'll be using two main technologies in this book: Python and GitHub Copilot.
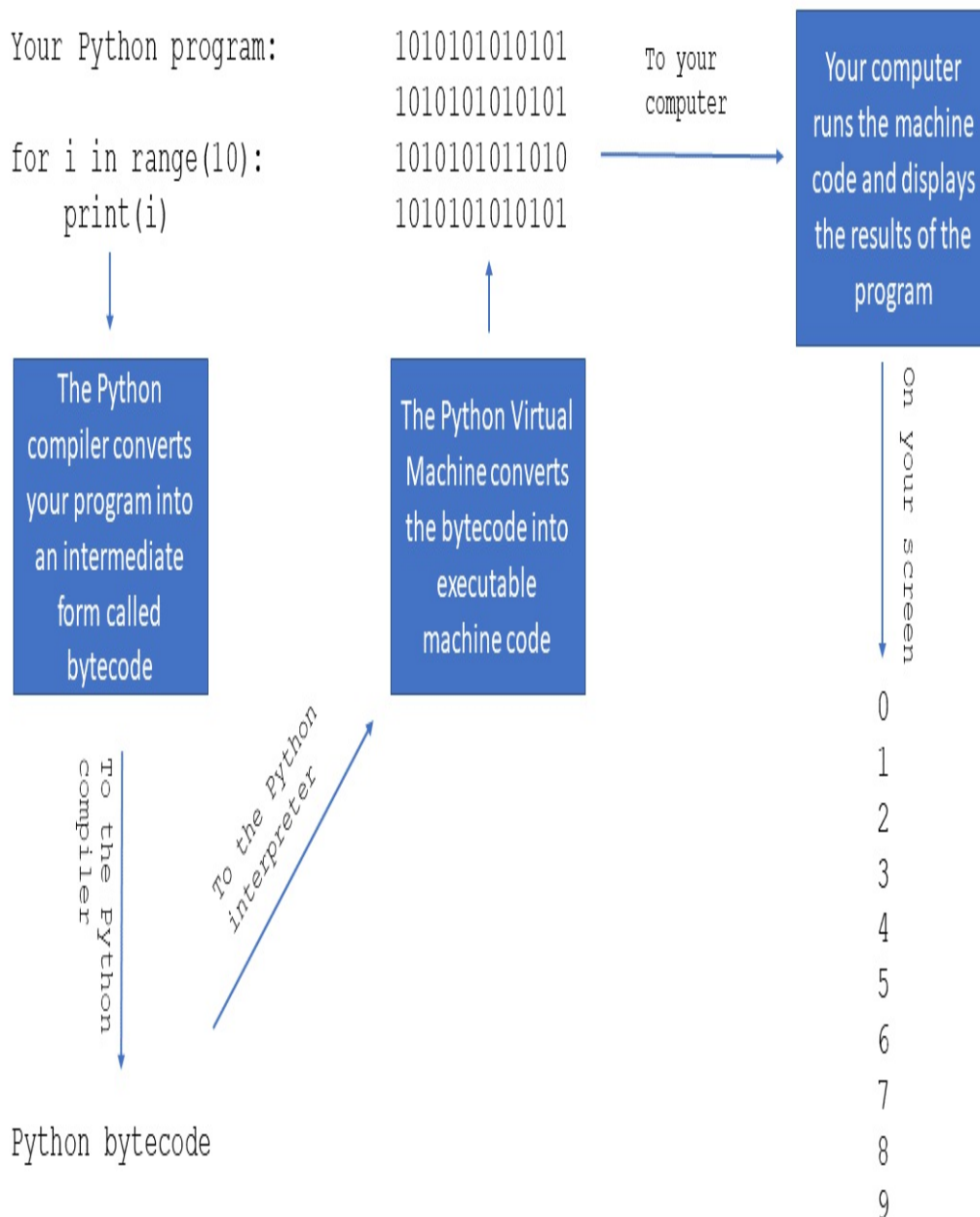
Python is a programming language. It's a way to communicate with a computer. People use it to write all kinds of programs that do useful things, like games, interactive websites, visualizations, apps for file organization, automating routine tasks, and so on.

There are other programming languages, too, like Java, C++, Rust, and many others. Copilot works with those, too, but at the time of writing, it works really well with Python. Python code is a lot easier to write compared to many other languages (especially assembly code). Even more importantly, Python is easy to *read*. After all, we're not going to be the one writing the Python code. Our AI assistant is!

Computers don't actually know how to read and run Python code. The only thing computers can understand is something called *machine code*, which looks even more ridiculous than assembly code as it is the binary

representation of the assembly code (yep, just a bunch of 0s and 1s!). Behind the scenes, your computer takes any Python code that you provide and converts it into machine code before it runs, as shown in Figure 1.1.

Figure 1.1 Your Python program goes through several steps before you see the output on your screen

```
Your Python program:          1010101010101      To your         Your computer
                              1010101010101      computer        runs the machine
for i in range(10):           1010101011010                      code and displays
    print(i)                  1010101010101                      the results of the
                                                                 program

        │                            ▲                                   │
        ▼                            │                                   │ On your screen
   The Python                  The Python Virtual                        │
   compiler converts           Machine converts                         ▼
   your program into           the bytecode into                        0
   an intermediate             executable                                1
   form called                 machine code                             2
   bytecode                                                             3
                                                                         4
        │                        ▲                                       5
        │ To the Python          │ To the Python                         6
        │ compiler               │ interpreter                           7
        ▼                                                                8
                                                                         9
   Python bytecode
```

Copilot, your AI Assistant

What is an *AI Assistant*? An AI Assistant is an Artificial Intelligence (AI)
agent that helps you get work done. Maybe you have an Amazon Alexa
device at home, or an iPhone with Siri—these are AI assistants. Those ones

help you order groceries, learn the weather, or determine that, yes, the woman who played Bellatrix in the Harry Potter movies really was in Fight Club. An AI assistant is just a computer program that responds to normal human inputs like speech and text with human-like answers.

Copilot is an AI Assistant with a specific job: it converts English into computer programs. (It can also do a whole lot more as we will soon see.) There are other AI assistants like Copilot, including CodeWhisperer, Tabnine, and Ghostwriter. We chose Copilot for this book by a combination of the quality of code that we have been able to produce, stability (it has never crashed for us!), and our own personal preferences. We encourage you to check out other tools as well when you feel comfortable doing so.

How Copilot works behind the scenes—in 30 seconds

You can think of Copilot as a layer between you and the computer program you're writing. Instead of writing the Python directly, you simply describe the program you want in words—this is called a *prompt*—and Copilot generates the program for you.

The brains behind Copilot is a fancy computer program called a **large language model**, or LLM. An LLM stores information about relationships between words, including which words make sense in certain contexts, and uses this to predict the best sequence of words to respond to a prompt.

Imagine that we asked you what the next word should be in this sentence: "The person opened the _____". There are many words that you could fill in here, like "door" or "box" or "conversation," but there are also many words that would not fit here, like "the" or "it" or "open." An LLM takes into account the current context of words to produce the next word, and it keeps doing this until it has completed the task.
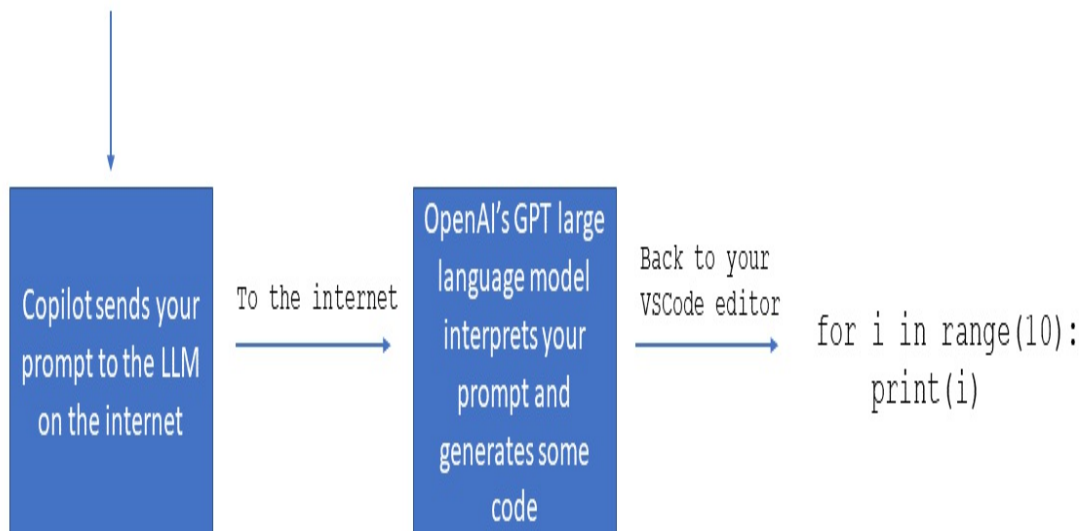
Notice that we didn't say anything about Copilot having an understanding of what it is doing. It just uses the current context to keep writing code. Keep this in mind throughout your journey: only we know whether the code that's generated actually does what we intended it to do. Very often it does, but you should always exercise healthy skepticism regardless.

Figure 1.2 will give you an idea of how Copilot goes from prompt to program.

Figure 1.2 Going from prompt to program with Copilot

```
You type a prompt like

# Output the numbers from 0 to 9
```

| Copilot sends your prompt to the LLM on the internet | To the internet → | OpenAI's GPT large language model interprets your prompt and generates some code | Back to your VSCode editor → | `for i in range(10):`
`    print(i)` |

You might wonder why Copilot writes Python code for us and not machine code directly. Isn't Python an expendable intermediate step now? Well, no, and the reason is that Copilot is going to make mistakes. And if it's going to make mistakes, and we need to get them fixed, then it's a lot easier to do that with Python than with machine code.

In fact, virtually no one checks if the machine code produced from Python is correct. This is partially because of the determinism of the Python language specification. One could imagine a future where Copilot conversations are so accurate that inspecting the Python is unnecessary, but we're a long way from that.

1.3 How Copilot changes how we learn to program

To illustrate how learning to program changes in the presence of an AI Assistant, we want to show you how powerful Copilot is at solving a programming task typically given to students in the middle of a programming course or in a book on how to program.

When learning how to program in the past, learners often spent most of their time learning how to work with the syntax and basic structure of programs. Being able to write a program from scratch (line-by-line), like the one we'll show you below, is seen as the culmination of all of their learning. Before we dive into the problem, we need to be clear: many new programmers cannot write the code to solve this problem despite the fact most programming instructors (and book authors) think that new programmers should be able to do so. Here are the instructions for one version of this problem [6].

"Write a program that will read the amount of rainfall for each day. A negative value of rainfall should be rejected since this is invalid and inadmissible. The program should print out the number of valid recorded days, the number of rainy days, the rainfall over the period, and the maximum amount of rain that fell on any one day. Use a sentinel value of 9999 to terminate the program."

If you have tried to learn to program before, this kind of problem statement may be familiar to you, and you may have struggled to solve such a problem.

But now, if we give exactly this prompt to Copilot, we get the following Python code as a result. (Remember from the Introduction that Copilot is nondeterministic, so if we ask it again, we might get different code!)

Listing 1.1 Copilot code that solves the classic rainfall problem.

```
rainfall = 0
total_rainfall = 0
num_rainy_days = 0
num_valid_days = 0
max_rainfall = 0
rainfall = int(input("Enter the amount of rainfall for the day: "
while rainfall != 9999:
    if rainfall < 0:
        print("Invalid rainfall amount. Please try again.")
        rainfall = int(input("Enter the amount of rainfall for t
    else:
        num_valid_days += 1
        total_rainfall += rainfall
        if rainfall > 0:
            num_rainy_days += 1
        if rainfall > max_rainfall:
            max_rainfall = rainfall
        rainfall = int(input("Enter the amount of rainfall for t
print("Number of valid days:", num_valid_days)
print("Number of rainy days:", num_rainy_days)
print("Total rainfall over the period:", total_rainfall)
print("Maximum amount of rain that fell on any one day:", max_rai
```

We're not expecting you to have any intuition right now that this code is good. But it is. As Computer Science Professors, we'd grade this code very highly.

People learning to program used to spend weeks or months to get to a point where they could write programs like this. Now Copilot can offer code immediately. As we'll see in the rest of the book, we still need to work to verify that this code is correct because Copilot can make mistakes. However, we don't need to write it from scratch anymore. We believe this successful interaction with Copilot signals the end of the way that we have historically taught and learned programming.

You, as someone interested in learning how to program, simply don't need to

struggle with syntax, control flow, and the host of other Python concepts needed to write code like this as you had to in the past. Sure, we are going to learn about those concepts in this book, but not so that you can demonstrate your understanding by writing code from scratch that Copilot can produce easily. No, we'll learn those concepts only inasmuch as they help us solve meaningful problems and interact productively with Copilot. **Instead, you get to learn how to write larger, more meaningful software faster, because of how an AI-Assistant fundamentally changes the skills needed to learn to program.**

1.4 What else can Copilot do for us?

As we've seen, we can use Copilot to write Python code for us starting from an English description of what we want. Programmers use the word *syntax* to refer to the symbols and words that are valid in a given language. So, we can say that Copilot takes a description in English syntax and gives us back code in Python syntax. That's a big win, because learning programming syntax has historically been a major stumbling block for new programmers. What kind of bracket— [, (, or { —am I supposed to use here? Do I need indentation here or not? What's the order that we're supposed to write these things: x and then y, or y and then x?

Such questions abound and let's be honest: it's uninteresting stuff. Who cares about this when all we want to do is write a program to make something happen? Copilot can help free us from the tedium of syntax. We see this as an important step to help more people successfully write programs, and we look forward to the day when this artificial barrier is completely removed. For now, we still need Python syntax, but at least Copilot helps us with it.

But that's not all Copilot can do. Here are some associated—and no less important—tasks where Copilot can help us:

- **Explaining code**. When Copilot generates Python code for us, we'll need to determine whether that code does what we want. Again, as we said above, Copilot is going to make mistakes. We're not interested in teaching you every nuance of how Python works (that's the old model of programming). We *are* going to teach you how to read Python code to

gain an overall understanding of what it does. But we're also going to use the feature of Copilot that explains code to you in English. When you finish with this book and our explanations, you'll still have Copilot available to help you understand that next bit of gnarly code that it gives you.

- **Making code easier to understand**. There are many different ways to write code to accomplish the same task. Some of them may be easier to understand than others. Copilot has a tool that can reorganize your code to make it easier for you to work with. For example, code that's easier to read is often easier to enhance or fix when needed.
- **Fixing bugs**. A *bug* is a mistake made when writing a program that can result in the program doing the wrong thing. Sometimes, you'll have some Python code, and it almost works, or works almost always but not in one specific circumstance. If you've listened to programmers talk, you may have heard the common story where a programmer would spend hours only to finally remove one = symbol that was making their program fail. Not a fun few hours! In these cases, you can try the Copilot feature that helps to automatically find and fix the bug in the program.

1.5 Risks and challenges of using Copilot

Now that we're all pumped up about getting Copilot to write code for us, we need to talk about the dangers inherent in using AI Assistants. See references [2] and [3] for elaboration on some of these points.

Copyright. As we discussed above, Copilot is trained on human-written code. More specifically, it was trained using millions of GitHub repositories containing open-source code. One worry is that Copilot will "steal" that code and give it to us. In our experience, Copilot doesn't often suggest a large chunk of someone else's code, but that possibility is there. Even if the code that Copilot gives us is a melding and transformation of various bits of other people's code, there may still be licensing issues. For example, who owns the code produced by Copilot? There is currently no consensus on the answer. The Copilot team is adding features to help; for example, Copilot will be able to tell you whether the code that it produced is similar to already-existing code and what the license is on that code [4]. Learning and experimenting on

your own is great, and we encourage that—but take the necessary care if you do intend to use this code for purposes beyond your home. We're a bit vague here, and that's intentional: it may take some time for laws to catch up to this new technology. It's best to play it safe while these debates are had within society.

Education. As instructors of introductory programming courses ourselves, we have seen first-hand how well Copilot does on the types of assignments we have historically given our students. In one study [5], Copilot was asked to solve 166 common introductory programming tasks. And how well did it do? On its first attempt, it solved almost 50% of these problems. Give Copilot a little more information, and that number goes up to 80%. You have already seen for yourself how Copilot solves a standard introductory programming problem. Education needs to change in light of tools like Copilot, and instructors are currently discussing how these changes may look. Will students be allowed to use Copilot, and in what ways? How can Copilot help students learn? And what will programming assignments look like now?

Code quality. We need to be careful not to trust Copilot, especially with sensitive code or code that needs to be secure. Code written for medical devices, for example, or code that handles sensitive user data must always be thoroughly understood. It's tempting to ask Copilot for code, marvel at the code that it produces, and accept that code without scrutiny. But that code might be plain wrong. In this book, we will be working on code that will not be deployed at large, so while we will focus on getting correct code, we will not worry about the implications of using this code for broader purposes. In this book, we start building the foundations that you will need to independently determine whether code is correct.

Code security. As with code quality, code security is absolutely not assured when we get code from Copilot. For example, if we were working with user data, getting code from Copilot is not enough. We would need to perform security audits and have expertise to determine that the code is secure. Again, though, we will not be using code from Copilot in real-world scenarios. Therefore, we will not be focusing on security concerns.

Not an expert. One of the markers of being an expert is awareness of what one knows and, equally importantly, what one doesn't. Experts are also often

able to state how confident they are in their response; and, if they are not confident enough, they will learn further until they know that they know. Copilot, and LLMs more generally, do not do this. You ask them a question, and they answer, plain as that. They will confabulate if necessary. They will mix bits of truth with bits of garbage into a plausible sounding but overall nonsensical response. For example, we have seen LLMs fabricate obituaries for people who are alive, which doesn't make any sense, yet the "obituaries" do contain elements of truth about people's lives. When asked why an abacus can perform math faster than a computer, we have seen LLMs come up with responses—something about abacuses being mechanical and therefore necessarily the fastest. There is ongoing work in this area for LLMs to be able to say, "sorry, no, I don't know this," but we are not there yet. They don't know what they don't know and that means they need supervision.

Bias. LLMs will reproduce the same biases present in the data on which they were trained. If you ask Copilot to generate a list of names, it will generate primarily English names. If you ask for a graph, it may produce a graph that doesn't consider perceptual differences among humans. And if you ask for code, it may produce code in a style reminiscent of how dominant groups write code. (After all, the dominant groups wrote most of the code in the world, and Copilot is trained on that code.) Computer science and software engineering have long suffered with a lack of diversity. We cannot afford to stifle diversity further, and indeed we need to reverse the trend. We need to let more people in and allow them to express themselves in their own ways. How this will be handled with tools like Copilot is currently being worked out and is of crucial importance for the future of programming. However, we believe Copilot has the potential to improve diversity by lowering barriers for entry into the field.

1.6 The skills we need

If Copilot can write our code, explain it, and fix bugs in it, are we just done? Do we just tell Copilot what to do and celebrate our pure awesomeness?

No. It's true that some of the skills that programmers rely upon (writing correct syntax, for example) will decrease in importance. But other skills remain critical. For example, you cannot throw a huge task at Copilot like,

"Make a video game. Oh, and make it fun." Copilot will fail. Instead, we need to break down such a large problem into smaller tasks that Copilot can help us with. And how do we break a problem down like that? Not easily, it turns out. This is a key skill that humans need to hone in their conversations with tools like Copilot, and a skill that we will teach throughout the book.

Other skills, believe it or not, may take on even more importance with Copilot than without. Testing code has always been a critical task in writing code that works. We know a lot about testing code written by humans, because we know where to look for typical problems. We know that humans often make programming errors at the boundaries of values. For example, if we wrote a program to multiply two numbers, it's likely that we'd get it right with most values but maybe not for when one value is 0. What about code written by AI, where twenty lines of flawless code could hide one line so absurd that we likely wouldn't expect it there? We don't have experience with that. We need to test even more carefully than before.

Finally, some required skills are entirely new. The main one here is called *prompt engineering*, which involves how to tell Copilot what to do. When we're asking Copilot to write some code, we're using a *prompt* to make the request. It's true that we can use English to write that prompt and ask for what we want, but that alone isn't enough. We need to be very precise if we want Copilot to have any chance of doing the right thing. And even when we are precise, Copilot may still do the wrong thing. In that case, we need to first identify that Copilot has indeed made a mistake, and then tweak our description to hopefully nudge it in the right direction. In our experience, seemingly minor changes to the prompt can have outsized effects on what Copilot produces.

In this book, we will teach you all of these skills.

1.7 Societal concerns about AI code assistants like Copilot

There's a lot of societal uncertainty right now about AI code assistants like Copilot. We thought we'd end the chapter with a few questions and our current answers. Perhaps you've been wondering about some of these

questions yourself! Our answers may turn out to be hilariously incorrect, but they do capture our current thoughts as two professors and researchers who have dedicated their careers to teaching programming.

Q: Are there going to be fewer tech and programming jobs now that we have Copilot?

A: Probably not. What we do expect to change is the nature of these jobs. For example, we see Copilot as being able to help with many tasks typically associated with entry-level programming jobs. This doesn't mean that entry-level programming jobs go away, only that they change as programmers are able to get more done given increasingly sophisticated tools.

Q: Will Copilot stifle human creativity? Will it just keep swirling around and recycling the same code that humans have already written, limiting introduction of new ideas?

A: We suspect not. Copilot helps us work at a higher level, further removed from the underlying machine code, assembly code, or Python code. Computer scientists use the term *abstraction* to refer to the extent that we can disconnect ourselves from low level details of computers. Abstraction has been happening since the dawn of computer science, and we don't seem to have suffered for it. On the contrary, it enables us to ignore problems that have already been solved and focus on solving broader and broader problems. Indeed, it's been the advent of better programming languages that have facilitated better software – software that powers Google search, Amazon shopping carts, and Mac OS weren't written (and likely could not have been written) when we only had assembly!

Q: I keep hearing about ChatGPT. What is it? Is it the same as Copilot?

A: It's not the same as Copilot, but it's built on the same technology. Rather than focus on code, though, ChatGPT focuses on knowledge in general. And as a result, it has insinuated itself into a wider variety of tasks than Copilot. For example, it can answer questions, write essays, and even do well on a Wharton MBA exam [7]. Education will need to change as a result: we cannot have people ChatGPT'ing their ways to MBAs! The ways in which we spend our time in worthwhile ways may change. Will humans keep

writing books, and in what ways? Will people want to read books knowing that that they were partially or fully written by AI? There will be impacts across industries, including finance, health care, and publishing [8]. At the same time, there is unfettered hype right now, so it can be difficult to separate truth from fiction. This problem is compounded by the simple truth that no one knows what's going to happen here long-term. In fact, there's an old adage coined by Roy Amara (known as Amara's Law) that says, "We tend to overestimate the effect of a technology in the short run and underestimate the effect in the long run." As such, we need to do our best to be tuned into the discussion so that we can adapt accordingly.

In the next chapter, we'll get you started with using Copilot on your computer so you can get up and running writing software.

1.8 Summary

- Copilot is an AI Assistant, which is an Artificial Intelligence (AI) agent that helps you get work done.
- Copilot changes how humans interact with computers, and the way that we write programs.
- Copilot changes the focus of skills we need to hone (less focus on syntax, more focus on prompt engineering and testing).
- Copilot is nondeterministic; sometimes it produces correct code, sometimes it doesn't. We need to be vigilant.
- Issues around copyright of code, education and job training, and bias in Copilot results still need to be worked out.

1.9 References

[1] G. Heyman, R. Huysegems, P. Justen, and T. Van Cutsem. "Natural language-guided programming," In 2021 Proc. ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software, Oct. 2021, pp. 39-55.

[2] N. A. Ernst and G. Bavota. "AI-Driven Development Is Here: Should You Worry?" IEEExplore.iee.org.

https://ieeexplore.ieee.org/document/9713901/figures#figures (Accessed Feb. 7, 2023.)

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.D.O. Pinto, J. Kaplan, ... and W. Zaremba. "Evaluating large language models trained on code," 2021. arXiv preprint arXiv:2107.03374. https://arxiv.org/abs/2107.03374

[4] R. D. Caballar. "Ownership of AI-Generated Code Hotly Disputed > A copyright storm may be brewing for GitHub Copilot." Spectrum.IEEE.org. https://spectrum.ieee.org/ai-code-generation-ownership (Accessed Feb. 7, 2023.)

[5] P. Denny, V. Kumar, and N. Giacaman. "Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language," 2022. arXiv preprint arXiv:2210.15157.

[6] A. Ebrahimi, "Novice programmer errors: language constructs and plan composition," Int. J. Hum.-Comput. Stud. 41, 4 (Oct. 1994), pp. 457–480.

[7] A. Zilber. "AI bot ChatGPT outperforms students on Wharton MBA exam: professor." NYPost.com. https://nypost.com/2023/01/23/chatgpt-outperforms-humans-on-wharton-mba-exam-professor/ (Accessed Feb. 7, 2023.)

[8] A. Mitchell. "ChatGPT could make these jobs obsolete: 'The wolf is at the door'." NYPost.com. https://nypost.com/2023/01/25/chat-gpt-could-make-these-jobs-obsolete/ (Accessed Feb. 7, 2023.)

[1] Based on code from: https://draftsbook.com/part-7-conditions-and-loop-uses-in-assembly-language/

2 Getting Started with Copilot

This chapter covers

- Setting up Python, VSCode, and Copilot on your personal system
- Introduction to the design process with Copilot by writing (mostly) human language
- Copilot's value in writing code for a basic data processing task

This chapter will help you get started working with Copilot on your own machine and will familiarize you with how to interact with Copilot. After getting you setup with Copilot, we'll ask that you follow along with our examples when you can. There's no substitute for practice and we believe you'll be able to learn right alongside us for the remainder of the book.

Once you've set up Copilot, we'll walk through a fun example with Copilot that showcases the power of Copilot in solving standard tasks. As we walk through that example, you'll see how we're interacting with Copilot and how we're able to write software without writing any actual code ourselves.

2.1 Time to setup your computer to keep learning

Learning how to write software requires that you perform the task of actually writing software, not just reading about it. If this were a book on how to play guitar, would you keep reading it without ever trying to play the guitar? We thought not. Reading this book without following along and trying it out yourself would be like watching a marathon runner finish the race and thinking you're ready to go run one yourself. We'll stop with the analogies, but seriously, you need to get your software installed and running before we go farther.

What scares us the most right now is that we just hit the most common point where novices, even those eager to learn programming, tend to fail, and we *really* want to see you succeed. Now, you might be thinking, "Wait, really?

We're just getting started." Yes, that's exactly the point. In Leo's popular Coursera course about learning Java programming [1], can you guess the point when most new leaners leave? Is it the challenging assignment that involves plotting earthquake markers on the globe in real time at the end of the course? No. It's actually the warmup assignment where the learners must set up their programming environment. As such, we understand this could be a hurdle for you. We hope that with this not-so-subtle nudge, we can help you achieve all the goals you had in mind when you bought this book. It all starts with installing the software.

2.1.1 Overview of the software in your programming environment

To set up and use Copilot easily, we'll be installing the software editing tools used by novices and software engineers alike. The tools you will be using are GitHub, Copilot, Python, and Visual Studio Code. Of course, if you already have all these tools installed, jump to section "Our first programming problem" later in this chapter.

GitHub Account

GitHub is an industry standard tool for developing, maintaining, and storing software. We won't be using GitHub in this book, however. We're signing up for GitHub simply because you'll need an account to get access to Copilot. Signing up for a GitHub account is free but, at the time of writing, they charge for Copilot. If you are a student, they will waive that fee. If you aren't a student, as of writing, you can get a 30-day free trial.

You might ask why they charge for the service, and there's a good answer. It's expensive to build the GPT3 models (imagine thousands of computers running for a year to build the model) and there are costs they incur by providing predictions from the model (many machines are receiving your input, running it through the model, and generating your output). If you are not ready to commit to using Copilot, you could make a calendar note for roughly 25 days from the day you sign up and if you aren't using Copilot at that time, just cancel. If, on the other hand, you have succeeded in learning how to write software with Copilot and are using it to improve your

productivity at work or just as a hobby, keep it.

Python

Really any programming language would have worked for this book, but we picked Python because it is one of the most popular programming languages in the world and is the language we teach in our introductory courses at our universities. As we said in Chapter 1, compared to other languages Python is easier to read, easier to understand, and easier to write. For this book, Copilot will primarily be generating the code, not you. However, you will want to be able to read and understand the code generated by Copilot and Python is great for that.

Visual Studio Code (VSCode)

You can use any text editor to program. However, if you want a nice programming environment where you can write code, easily get suggestions from Copilot, and run your code, VSCode is our preferred tool. VSCode is used by novices learning software and is well liked by students [2]. It's also used globally by professional software engineers, which means you'll be able to work and learn while using this environment after finishing the book.

For VSCode to work for this book, you'll need to install a few extensions that enable working with Python and using Copilot, but one of the great things about VSCode is that it makes it easy to install those extensions.

2.2 Getting your system setup

This is actually a four-step process. To streamline this chapter, we're just providing the main steps for this process below. However, there are more detailed instructions available in the following locations:

- GitHub's documentation: https://docs.github.com/en/copilot/getting-started-with-github-copilot
- In the appendix of this book, there are detailed instructions for setting up your system for both PC and MacOS. As these websites might change

after we write this book, we encourage you to use a combination of the link above and the appendix together.

- In the online book forum, you can ask for help and see a list of answers to frequently asked questions.

The primary steps you'll need to accomplish are as follows:

1. **Set up your GitHub account and sign up for Copilot**
   a. Go to GitHub.com and sign up for a GitHub account: https://github.com/signup
   b. Go into your settings in GitHub and enable Copilot. This is the point where you'll either need to verify you are a student or sign up for the 30-day free trial (available at the time of writing).
2. **Install Python**
   a. Go to the website, www.Python.org/downloads/
   b. Download and install the latest version of Python (3.11.1 at the time of writing).
3. **Install Visual Studio Code**
   a. Go to the website, https://code.visualstudio.com/Download, Select the main download for your Operating System (e.g., Windows Download or Mac Download).
   b. Download and install the latest version of Visual Studio Code.
4. **Install Visual Studio Code Extensions (please read through https://code.visualstudio.com/docs/editor/extension-marketplace for details)**
   a. Python (by Microsoft). Follow the instructions at https://code.visualstudio.com/docs/languages/python to set up the Python extension correctly (specifically selecting the correct interpreter).
   b. GitHub Copilot (by GitHub)
   c. GitHub Copilot Labs (by GitHub). Note that Copilot Labs is not needed for the majority of the book, so if you encounter a waiting list or it has changed from the time of writing, please do not be discouraged.

We know that the steps here are brief, if you encounter any issues, we encourage you to consult the resources mentioned earlier for more detailed

setup instructions.

2.3 Working with Copilot in Visual Studio Code

Now that you have your system set up, let's get acquainted with the VSCode interface shown in figure 2.1. (You may need to click on the "Explorer" icon in the middle/top left to have this same view.) The following regions are identified in figure 2.1:

- Activity Bar - on the far left is the activity bar where we can open file folders or install extensions (as you did to install the Github Copilot extension in the previous section).
- Side Bar – shows what is presently open in the Activity Bar. In figure 2.1, the Activity Bar has the "Explorer" selected and hence the Side Bar is showing the files in the present folder.
- Editor Pane(s) - these are the primary areas we will be using to create our software. The editor in the Editor Pane is similar to any other text editor in that you can write text, edit text, copy and paste using the clipboard, and so on. What is special about it, however, is that it is designed to work well with code. As we'll see in the next example, you will be primarily working in this window by asking Copilot to generate code and then testing that code.
- Output and Terminal Panel – this is the area of the interface for seeing the output of your code or any errors that have occurred. It has the tabs "Problems", "Output", "Debug Console", and "Terminal." We will primarily be using the "Problems" tab where we can see potential errors in our code and the "Terminal" tab that allows us to interact with Python and will be where we see the output of our code.

We highlighted the Copilot logo in the bottom right of figure 2.1 as you should be seeing this symbol (or similar) if you setup Copilot properly in the previous section.

Figure 2.1 The VSCode Interface [3].
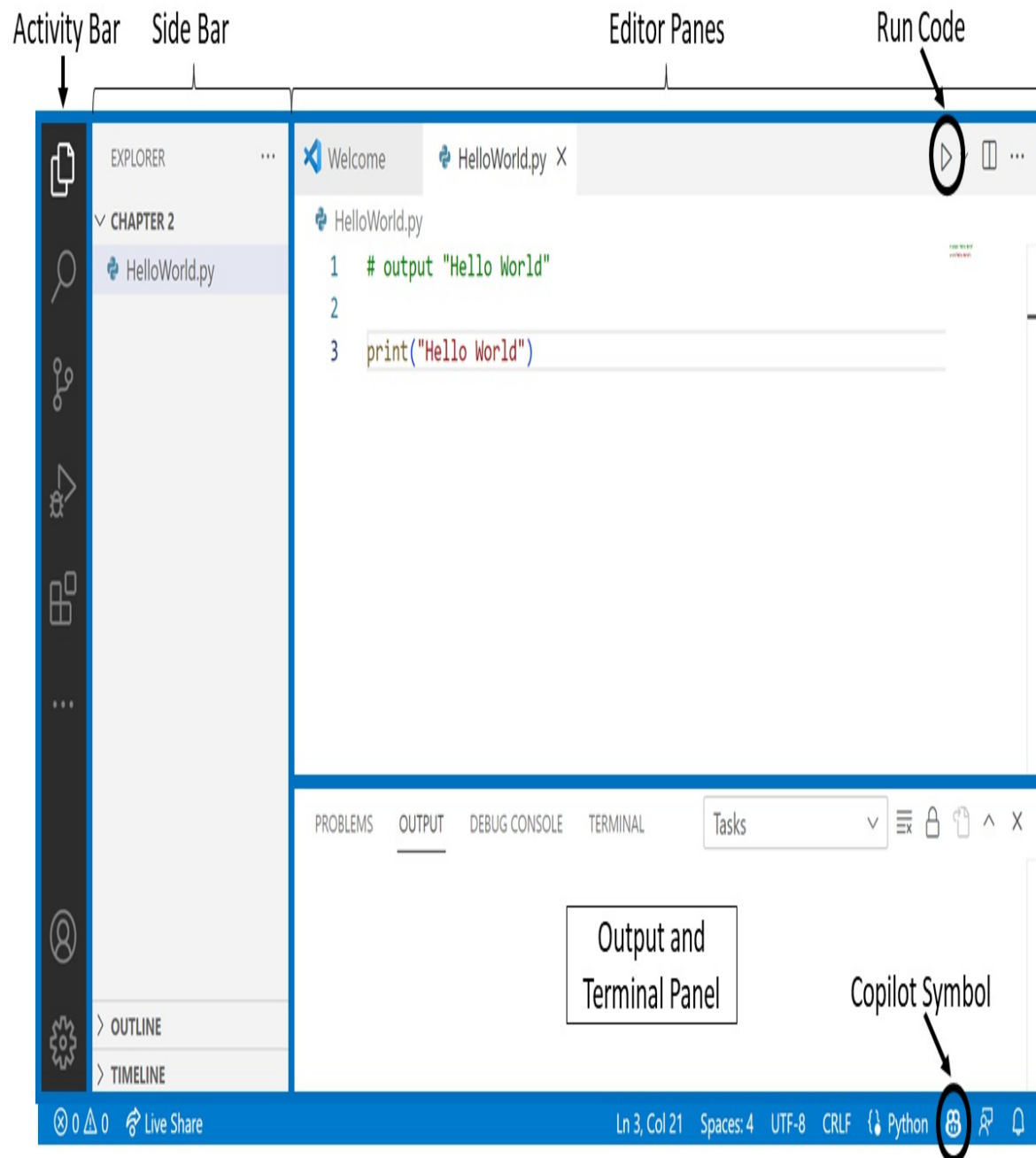
Set up your working folder

In the top of the Activity Bar on the left in VSCode you will find the "Explorer" as the top icon. After you click on the "Explorer", it should say

"No Folder Open." Click on the button to "Open Folder" and select a folder in your computer (or make a new one—we like the folder name fun_with_Copilot). Once you've opened this folder, your workspace will be the folder you opened which means you should have your code and any data files, like the one we'll use later this chapter, in that folder.

File not found or file missing errors

If you ever receive an error that says you are missing a file, these are the kind of errors that can be really annoying when writing software. It could be that you just didn't put the file in your working folder, this happens. That's an easy fix by copying or moving the file into the correct folder. However, sometimes, you'll look in the folder and the file will be there, but when you run your code in VSCode, Python can't seem to find it. If this happens to you (it happened to us when writing the book!), be sure to have the folder with the code and the desired file open using Explorer in VSCode (as shown in the Side Bar in figure 2.1).

Check to see if your setup is working properly

Let's check to see if we've set up everything properly and that Copilot is working. To do this, start by creating a new file to hold our program. You do this by going to "File->New File" (figure 2.2), then selecting "Python File" (figure 2.3).

Figure 2.2 How to create a new file in VSCode.
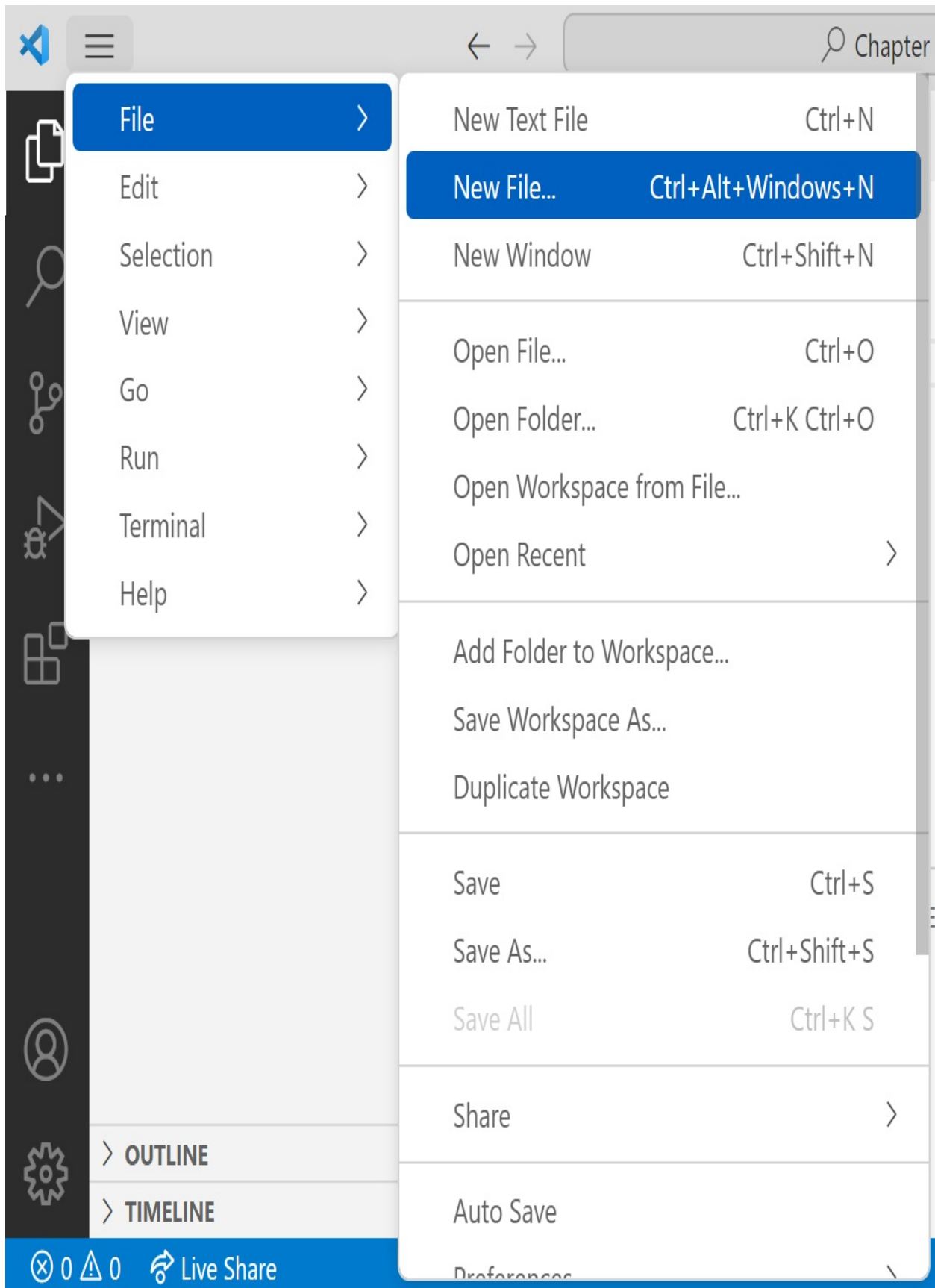
Figure 2.3 Select to create the New File as a Python File.
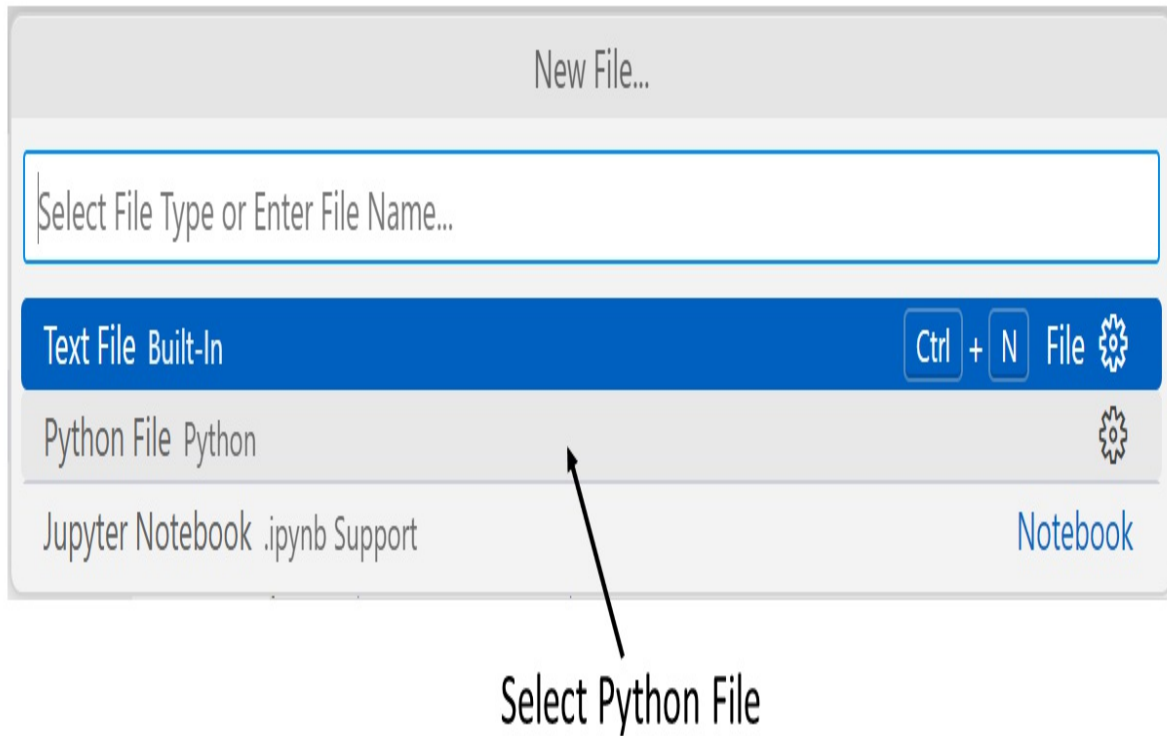
After creating it, we like to make sure that we've saved the file. Go to "File->Save As" and let's just name this file "first_Copilot_program".

Next, in the text editor, type:

```
# output "Hello Copilot" to the screen
```

The prompts and code we write will be in bold font (as above) to help distinguish between what we write and the code and comments Copilot may give us. The "#" sign at the start is important (and you should include it in what you typed). It means that what you wrote is a comment (depending on your VSCode color palette, it'll be a different color than the code we're about to produce). Comments are *not* code: the computer executes code and does not execute comments. Comments are used by programmers to provide a human-readable summary of what the code did to help other software engineers read the code. Today, its purpose has expanded to also work as a way to prompt Copilot. After writing a comment (and sometimes even while writing comments), Copilot will attempt to give us suggestions. You can think of this as a much more sophisticated autocomplete, like when you type "New York T" in your search engine, and it autocompletes with "New York Times."

To trigger Copilot to start giving us code (or more comments), press Enter at the end of the line, and you'll be at the start of a new line. Pause for a moment, and you should see something appear. Until accepted, Copilot's suggestions are in light gray italics. If you do not get a suggestion yet, you may need to hit enter a second time to trigger Copilot to suggest the code. Here's what happened for us:

```
# output "Hello Copilot" to the screen
print("Hello Copilot")
```

If you still do not see a suggestion from Copilot, try pressing <CTRL>+Enter (hold "CTRL" while pressing "Enter"). When you press <Ctrl>+Enter, a new window on the right-hand side of the screen should appear. The window will be to the right of your editor window with the program and will be called "GitHub Copilot". If that window does not appear, there may be something wrong with your setup and we encourage you to go to the Appendix to double check that you followed all the steps correctly or go to our book website to find (or ask for) help.

If you saw the suggestion, like above, from Copilot, then you need to just

press <Tab> to accept Copilot's suggestion. Once you do this, the suggestion that was previously in light gray italics should now be in a standard font like below:

```
# output "Hello Copilot" to the screen #A
print("Hello Copilot") #B
```

If you are seeing different code than this, it's because of something we mentioned in the introduction: Copilot is non-deterministic so you may see different code than us. We mention this because sometimes Copilot makes a minor mistake with the code here and may give you code similar to this:

```
print "Hello Copilot"
```

You might think this slight difference (no parentheses around "Hello Copilot") wouldn't matter, but it does. Before Python 3, this was the correct syntax for a print statement and when Python 3 was introduced, it switched to the code with parentheses. Since we're running Python 3, you need to have the parentheses for the code to work. You might ask why Copilot gets this wrong, but the issue is Copilot was trained on Python code before Python 3. If this seems annoying, we agree. But it's a hint of the frustration novice programmers went through before Copilot. Most of what Copilot suggests is syntactically correct. But if you were a novice writing the code from scratch, missing parentheses or a missing colon somewhere might cost you a lot of time.

Now that we have the correct code:

```
# output "Hello Copilot" to the screen
print("Hello Copilot")
```

which, as you might guess, prints "Hello Copilot" to the screen, we should test it. First, you'll want to save your file by going to File->Save.

Be sure to save your file before you run it

Be sure to always save your file before running it. We're embarrassed to admit the amount of time we've spent trying to fix code that was actually correct but hadn't been saved.

To run your program, go to the top right corner of the text editor and press the "Run Code" icon as shown in figure 2.1. After pressing the icon, in the Terminal section at the bottom, you should see something like this:

```
> & C:/Users/YOURNAME/AppData/Local/Programs/Python/Python311/Pyt
Hello Copilot
```

The top line starting with ">" is the command for the computer to run your code and all it says is to run your first_Copilot_program.py using Python. The second line is the output from running the command and it says "Hello Copilot" which is what we'd hoped to see.

Congratulations! You've written your first program! We now know that your programming environment is setup correctly and we can move onto our first programming task. But before we do, we'd like to go over some tips for how to deal with some common issues we've encountered when working with Copilot so you have these tips available to you when working through the next example.

2.4 Addressing common Copilot challenges

It may seem early in your experience with Copilot to start talking about common challenges with Copilot, but you may have already run into this when writing your first program and you'll certainly encounter some of these working through our next example and in the next chapter, so we wanted to give these to you now.

In our time working with Copilot, we've run into a few common challenges. These challenges will likely decrease with time as Copilot improves, but they were still issues at the time of writing. Although the challenges in table 2.1 are not exhaustive of what you might encounter, we hope our tips on how to address these common challenges will help you get up and running quickly. We'll keep a running list at our book's website so please feel free to reach out to us on the forums if you feel we've missed something!

Table 2.1 Common Challenges working with Copilot

Challenge	Description	Remedies
Comments only	If you give Copilot a prompt using the comment symbol (#), when you start a new line, it wants to just give you more comments rather than code. For example: **# output "Hello Copilot" to the screen** `# print "Hello world to the screen` We've seen Copilot generate line after line of comments, sometimes repeating itself! When this happens, suggestion #3 (use docstrings) is sometimes the most effective.	1. Add a newline (Enter) between your comment and Copilot's suggestion to help it switch from comments to code. 2. If a newline doesn't work, you can type a letter or two of code (no comment symbol). A couple letters from a keyword in your prompt usually works. For example: **# output "Hello Copilot" to the screen** `pr` A couple letters from a keyword typically causes Copilot to give a code suggestion 3. Switch from using # comments to docstring comments like this: *"""* *output "Hello Copilot" to the screen*

		""" 4. Use <Ctrl>+Enter to see if Copilot will give you suggestions that are code rather than comments

Table 2.1 Common Challenges working with Copilot (continued)

Challenge	Description	Remedies
Wrong Code	Sometimes Copilot just gives you obviously wrong code from the start. (You'll learn throughout this book how to identify incorrect code!) In addition, sometimes Copilot seems to get stuck down wrong paths. For example, it might seem to be trying to solve a different problem than	Much of this book is about how to address this issue, but here are some quick tips to get Copilot to help. 1. Change your prompt to see if you can describe what you need better. 2. Try using <Ctrl> + Enter to find a suggestion from Copilot that is the correct code. 3. Close the VSCode program, wait a little bit, and restart it. This can help clear the Copilot cache to get new suggestions. 4. Try breaking down the problem into smaller steps

	what you've asked it to solve. (Suggestion 3, in particular, can help with getting Copilot to go down a new path.)	(see Chapter 7 for more details). 5. Debug the code (see Chapter 8). 6. Try asking ChatGPT for the code and feed its suggestions into VSCode. A different LLM can sometimes give suggestions that help the other LLM to get unstuck.

Table 2.1 Common Challenges working with Copilot (continued)

Challenge	Description	Remedies
Copilot gives you:	We've had Copilot seem to tell us to write our own code by generating	We believe this is happening when we ask Copilot to solve a problem that has been given by an instructor to students to solve in the past. Why? Well, when we write our assignments for our students, we (as instructors) often write some code and then tell our students to write the rest by writing: `# YOUR CODE HERE`

`# YOUR CODE HERE`	this (or similar text) after a prompt: `# YOUR CODE HERE`	Where we want students to write their code. Students tend to leave that comment in their solution code which means Copilot was trained to think this comment is an important part of the solution (it's not). Often, we're able to solve this problem by finding reasonable solutions in the Copilot suggestions with <CTRL>+Enter, but please see the solutions for Wrong Code if that doesn't work.
Missing Modules	Copilot gives you code, but it won't work because there are modules missing. (Modules are additional libraries that can be added to Python to give pre-built functionality.)	See Section 2.5, under "Modules" for how to install new modules on your machine.

2.5 Our first programming problem

The goal of this next section is twofold: 1) for you to see the workflow of interacting with Copilot and 2) for you to gain an appreciation of how powerful Copilot can be by seeing it solve a fairly complicated task fairly easily.

In our next chapter, we'll talk through the workflow with Copilot in more

detail, but generally speaking, you'll use the following steps when authoring code with Copilot:

1. Write a prompt to Copilot using comments (#) or docstrings (""").
2. Let Copilot generate code for you
3. Check to see if the code is correct, by reading through it and by testing
   a. If it works, move to Step #1 for the next thing you'd like it to do.
   b. If it doesn't work, delete the code from Copilot and go back to Step #1 and modify the prompt (and see suggestions in table 2.1).

Because you've just started working with Copilot, we're wary of showing you such a large example, but we feel you'll really value seeing how powerful Copilot can be now that you have it installed. **As such, we want you to follow along as best as you can to get a feel for working with Copilot on your own, but if you get stuck, feel free to just read along and save working along with Copilot in VSCode for the next chapter.** Later chapters will explain the process of working with Copilot in more detail. Also, Copilot will generate a lot of code in this section and we don't expect you to understand the code until much later in the book. We provide the code solely so you can see what Copilot gave us, but do NOT feel as though you need to try to understand the code in this chapter.

To get started, let's create a new file. If you aren't already in VSCode, go ahead and start it. Then create a new Python file and save it as "nfl_stats".

Showcasing Copilot's value in a data processing Task

We want to start with some basic data processing as this is something that many of you have likely done in your personal or professional lives. To find a dataset, we went to a great website called Kaggle [4] which has tons of datasets freely available for use. Many of them include important data like health statistics for different countries, or information to help track the spread of disease, and so on. We're not going to use those because we'd like to have something lighter for our first program. Since both of us are American Football fans, we felt we should play with the NFL offensive stats database.

Let's get started by downloading the dataset from:

To download the dataset, you will have to sign up for a Kaggle account. If you don't want to create the account, it's okay to just read through this section without using VSCode and Copilot to generate the code yourself. Once downloaded, you may need to extract the zip file using the default zip extractor on your computer. Copy the dataset file from that zip file into your current folder in VSCode where you have your code (the folder you have open in Explorer). (If you are on a Mac and the file is saved as a .numbers file, you will need to use "File->Export To" to then save the file as a CSV in your current working directory.) That dataset has NFL information from 2019 to 2022.

Figure 2.4 The first few columns and rows of the nfl_offensive_stats.csv dataset.

```
   nfl_offensive_stats.csv  ✕

   nfl_offensive_stats.csv

   1    game_id,player_id,position ,player,team,pass_cmp,pass_att,pass_yds,
   2    201909050chi,RodgAa00,QB,Aaron Rodgers,GNB,18,30,203,1,0,5,37,47,91
   3    201909050chi,JoneAa00,RB,Aaron Jones,GNB,0,0,0,0,0,0,0,0,0,13,39,0,9
   4    201909050chi,ValdMa00,WR,Marquez Valdes-Scantling,GNB,0,0,0,0,0,0,0
   5    201909050chi,AdamDa01,WR,Davante Adams,GNB,0,0,0,0,0,0,0,0,0,0,0,0,0
   6    201909050chi,GrahJi00,TE,Jimmy Graham,GNB,0,0,0,0,0,0,0,0,0,0,0,0,0
   7    201909050chi,DaviTr03,WR,Trevor Davis,GNB,0,0,0,0,0,0,0,0,0,0,0,0,0
   8    201909050chi,TonyRo00,TE,Robert Tonyan,GNB,0,0,0,0,0,0,0,0,0,0,0,0
   9    201909050chi,WillJa06,RB,Jamaal Williams,GNB,0,0,0,0,0,0,0,0,5,0,0
   10   201909050chi,LewiMa00,TE,Marcedes Lewis,GNB,0,0,0,0,0,0,0,0,0,0,0
   11   201909050chi,TrubMi00,QB,Mitchell Trubisky,CHI,26,45,228,0,1,5,20,27
   12   201909050chi,DaviMi01,RB,Mike Davis,CHI,0,0,0,0,0,0,0,0,5,19,0,8,7
   13   201909050chi,MontDa01,RB,David Montgomery,CHI,0,0,0,0,0,0,0,0,6,18
```

The nfl_offensive_stats.csv file is something known as a comma separated value text file (see figure 2.4 for a portion of the file). This is a pretty standard format for storing data. It has a header row at the top that explains what's in every column. The way that we (or a computer) know the boundaries between columns is to use the commas between cells. Also notice that each row is placed on its own line.

Good news: Python has a bunch of tools for reading in CSV files.

Step 1: How many passing yards did Aaron Rodgers throw in 2019-2022

Let's start by exploring what is stored in this file. To preview what is in the file, you can look at the Kaggle webpage for these stats under "Detail", you can open it in VSCode, or you can just open it up in spreadsheet software like Microsoft Excel. (If you open it with Excel, be sure not to save the file. We need to leave the file in a .csv format.) Whichever way you choose to open it, here's the start of the header (top) row (also shown in figure 2.4):

```
game_id,player_id,position ,player,team,pass_cmp,pass_att,pass_yd
```

There are a bunch more columns after these, but these have all we need to perform our first task. We know now that there's a column for players and a column for passing yards. Aaron Rodgers is a player that gets passing yards in each game that he plays. But how many passing yards does he have in total, over all of the games that he played? This isn't so easy to answer by directly looking at the file. So what we want the computer to do is to make this easier for us!

We want it to sum up all the passing yards (pass_yds) for rows (games) where Aaron Rodgers is the player. For now, we're going to just ask for all the yards in the database even though it covers multiple seasons. We could change this later if we'd like. This problem might be a good problem to give to programmers learning to program in their 4th week of a standard college-level introductory programming course, but we have Copilot! So instead of learning how to write this code from scratch, we're just going to ask Copilot to generate it for us. To make that happen, we are going to be quite specific in our request to make sure Copilot knows what we are asking for. Specifically, we're only going to ask it to perform small amounts of work and then re-prompt it to perform the next step. Later we'll discuss how to write good prompts, but for now, just go ahead and use what we've written below by placing this text at the top of your new file:

```
"""
open the csv file called "nfl_offensive_stats.csv" and read in
the csv data from the file
"""
```

The """ at the top and the bottom are surrounding something called a

docstring. Docstrings are an alternative way of commenting (similar to text starting with #). They are commonly used for describing functions (see Chapter 3 for details on functions), but we use them in this example to avoid Copilot continually generating comments (see the "Comments only" problem in table 2.1). Given this prompt, Copilot should start generating code. For us, it produced this block of code:

```
import csv
with open('nfl_offensive_stats.csv', 'r') as f: #A
    reader = csv.reader(f)
    nfl_data = list(reader)
```

First, for the purpose of reading this book, we want to remind you that the prompt is displayed differently than what Copilot produces. This is intentional so you can tell what we wrote (and you should write) and what Copilot wrote.

Second, the code produced by Copilot is quite reasonable. We don't expect you to be able to understand the code at this point in the book, but you can likely see the name of the file we wanted opened and some code about opening the file and reading in the file. Later in the book, we'll learn how to read through the code. For now, just keep following along.

Now that we have the data from the file, we're going to give it a new prompt to ask it to sum all the passing yards for Aaron Rodgers in this dataset. Because the computer doesn't know what football is or specifics like that Aaron Rodgers is a quarterback, our prompt is going to be quite specific. We'll teach you how to write prompts like this over the course of the book. Here is the new prompt:

```
"""
In the data we just read in, the fourth column is the player
and the 8th column is the passing yards.  Get the sum of
yards from column 8 where the 4th column value is
"Aaron Rodgers"
"""
```

Notice how we tell the computer which columns are for players, and which are for passing yards. That's to tell the computer how to interpret the data. Also, notice how we say specifically that we only want to sum the yards in

the case that the player's name is Aaron Rodgers. Again, we'll teach you how to write prompts like this as we move forward in the book. Given this prompt, Copilot then produced the code below.

```
passing_yards = 0
for row in nfl_data:
    if row[3] == 'Aaron Rodgers':
        passing_yards += int(row[7])
print(passing_yards)
```

Reminder: Copilot is non-deterministic

Remember from Chapter 1 that Copilot is nondeterministic so what Copilot gives you may not match what it gives us. This is going to be a challenge for the rest of the book: what do you do if you get a wrong result when we get a right result? We're actually fairly confident that Copilot will give you a correct answer here, but **if you get a wrong answer from Copilot, go ahead and just read the remainder of this section rather than working along with Copilot in VSCode**. We will absolutely give you all the tools you need to fix the code when Copilot gives you a wrong answer, but that skill will be taught over the remainder of the book so we don't want you to get stuck on this now. When we run this code (recall how to "Run Code" from figure 2.1), we get this result: `13852`

Which is the correct answer. (We double checked the answer, but if you are familiar with football, you can likely use estimates to see if the figure seems reasonable. Quarterbacks throw for 3,000-5,500 yards per season, and this is three seasons worth of data, so 13,852 seems like the right ballpark for a high performing quarterback.) What's particularly interesting is that we planned on giving Copilot a third prompt to ask it to print the result, but Copilot guessed that's what we'd want to do so it did so on its own.

What do we want you to take from this example (and the rest of the chapter)?

1. **Copilot is a powerful tool.** We didn't write any code, but we were able to get Copilot to generate the code needed to perform a basic analysis of the data. For readers who have used spreadsheets, you can probably think of a way to do this using spreadsheet applications like Excel, but it likely wouldn't be as easy as writing code like this. Even if you haven't

used spreadsheets before, you've got to admit that it's amazing that writing some basic, human readable prompts can produce correct code and output like this.

2. **Breaking problems into small tasks is important.** For this example, we tried writing this code with just a single large prompt (not shown) or by breaking it into two smaller tasks. The larger prompt was almost identical text, just as a single prompt. What we found was that Copilot would usually give us the right answer with the larger prompt but would sometimes make mistakes. This was especially true in the next example we'll show you. However, breaking the problem into smaller tasks significantly increased the likelihood of Copilot generating the right code. We'll see how to break down larger problems into smaller tasks throughout the remainder of this book as this is one of the most important skills you'll need. In fact, the next chapter helps you start understanding what are reasonable tasks to give to Copilot.

3. **We still need to understand code to some degree.** This is true for several reasons. One is that writing good prompts requires you to have a basic understanding of what computers know and what they don't. We can't just give a prompt to Copilot that says, "Give me the number of passing yards for Aaron Rodgers." Copilot likely wouldn't be able to figure out where the data is stored, the format of the data, which columns correspond to players and passing yards, or that Aaron Rodgers is a player. We had to spell that out to Copilot for it to be successful. Another reason has to do with determining whether code from Copilot is reasonable. When the two of us read the response from Copilot, we know how to read code so we can determine if the code produced by Copilot is reasonable. You'll need to be able to do this to some degree, which is why Chapters 4 and 5 are dedicated to reading code.

4. **Testing is important.** When programmers talk about testing, they're referring to the practice of making sure that their code works correctly, even in possibly unexpected circumstances. We didn't spend much time on this piece other than checking if Copilot's answer is plausible using estimates on just one data set, but in general we'll need to spend more time on testing as this is a critical part of the code writing process. It likely goes without saying, but errors in code range from embarrassing (if you tell your hard-core NFL fan friend the wrong number of passing yards for a player) to dangerous (if software in a car behaves incorrectly)

to costly (if businesses make decisions on wrong analyses). Concerningly, even after you've learned how to read code, we have first-hand experience to tell you that even if the code looks correct, it might not be! To address this, we have to test every piece of code created by Copilot to ensure it does what it should. We'll learn how to rigorously test Copilot's code in later chapters.

To showcase the power of Copilot, we're going to continue this example. Please feel free to follow along writing the prompts and running the code in Copilot or just reading along.

Step 2: How well did all the quarterbacks do over that time period?

Knowing how well Aaron Rodgers did is interesting. But a next and more meaningful step would be to compare his stats to other quarterbacks over that time period. We really only want to compare against other quarterbacks because they are the players whose job is to throw the ball. Sure, a running back might throw the ball once a season, and it's often fun when they do, but it's not really their job. To include only the quarterbacks, we need to go back to the data for a moment. The third column is for "position" and "QB" stands for Quarterback. As such, let's delete all the code Copilot gave us (everything after the end of our initial comments) and start over. We won't show each step like we did above, instead, here is the result of the entire interaction with Copilot with our prompts highlighted as separate from the code it gave us.

Listing 2.1 Copilot's code to analyze the top Quarterbacks

```
""" #A
open the csv file called "nfl_offensive_stats.csv" and #A
read in the csv data from the file #A
""" #A

# import the csv module #B
import csv

# open the csv file
with open('nfl_offensive_stats.csv', 'r') as f:
    # read the csv data
    data = list(csv.reader(f))
```

```
""" #C
the 3rd column in data is player position, the fourth column #C
is the player, and the 8th column is the passing yards. #C
For each player whose position in column 3 is "QB", #C
determine the sum of yards from column 8 #C
"""#C

# create a dictionary to hold the player name and passing yards
passing_yards = {}

# loop through the data
for row in data:
    # check if the player is a quarterback
    if row[2] == 'QB':
        # check if the player is already in the dictionary
        if row[3] in passing_yards:
            # add the passing yards to the existing value
            passing_yards[row[3]] += int(row[7])
        else:
            # add the player to the dictionary
            passing_yards[row[3]] = int(row[7])

""" #D
print the sum of the passing yards sorted by sum #D
of passing yards in descending order #D
""" #D
for player in sorted(passing_yards, key=passing_yards.get, revers
    print(player, passing_yards[player])
```

Notice that we gave Copilot three prompts. The first was to handle the input data, the second was to process the data, and the third was to output the response. This cycle of input data, process data, provide output is exceptionally common in programming tasks.

Looking at the results from Copilot, we have to point out that we've taught programming for years and this is pretty impressive. We might ask students to solve something like this on a final exam in our college-level classes and we suspect less than half the class would do it correctly. Without diving into too many details, Copilot chose a good way of storing the data by using a dictionary (not a normal dictionary like an English dictionary, but a way of storing data in Python) which is a good choice here and used a clever way of sorting the data to help in displaying the results.

Thinking of the results, here are the first 5 lines from the output if you run the

code:

```
Patrick Mahomes 16132
Tom Brady 15876
Aaron Rodgers 13852
Josh Allen 13758
Derek Carr 13271
```

If you follow football, these results should not be a surprise to you. Just to see how well Copilot can adapt to our wishes, let's try to make a minor change. Suppose that because Tom Brady is already recognized as one of the best QBs of all time, you would rather omit him from this comparison.

To make this change, we're just going to modify the prompt at the bottom. Go to the point in the code where it says:

```
"""
print the sum of the passing yards sorted by sum
of passing yards in descending order
"""
for player in sorted(passing_yards, key=passing_yards.get, revers
    print(player, passing_yards[player])
```

Delete the code, just leaving the comment, and add another line to the docstring like this:

```
"""
print the sum of the passing yards sorted by sum
of passing yards in descending order
Do not include Tom Brady because he wins too much
"""
```

Copilot then suggested to us:

```
for player in sorted(passing_yards, key=passing_yards.get, revers
    if player != "Tom Brady": #A
        print(player, passing_yards[player])
```

That's exactly what we'd like to see changed in the code. (Thanks, Tom Brady, for being a good sport in this example.) The code excluded all data for Tom Brady at the point of printing the results. When we save the file and run it again, the first 5 lines are now:

```
Patrick Mahomes 16132
Aaron Rodgers 13852
Josh Allen 13758
Derek Carr 13271
Matt Ryan 13015
```

Step 3: Let's Plot these stats so we can compare them better

Let's really drive home our key point that Copilot is a powerful tool by asking it to go even one step farther. The printout of all the quarterback stats is likely a useful analysis for some purposes. But a visual plot might be a better way of presenting this information. Can we ask Copilot to plot it? Suppose we just care about the top performing quarterbacks and so we decide to set an arbitrary limit of 4,000 yards as the minimum number of yards a quarterback needs to throw during this time window to be included in our plot. Granted, asking Copilot to plot the data is likely to introduce some new challenges because it will involve going beyond straight text, but let's give it a try.

At the end of the code we just wrote, add the following comment:

```
"""
plot the players by their number of passing yards only for
players with more than 4000 passing yards
"""
```

When we gave Copilot this prompt, here is the code it gave us:

Listing 2.2 Code given by Copilot to plot our data about quarterbacks.

```
import matplotlib.pyplot as plt
import numpy as np
# create a list of players
players = []
# create a list of passing yards
yards = []
# for each player in the dictionary
for player in passing_yards:
    # if the player has more than 4000 passing yards
    if passing_yards[player] > 4000:
        # add the player to the list of players
        players.append(player)
```

```
        # add the passing yards to the list of passing yards
        yards.append(passing_yards[player])

# create a numpy array of the passing yards
yards = np.array(yards)
# create a numpy array of the players
players = np.array(players)
# sort the players by the passing yards
print(np.argsort(yards))
players = players[np.argsort(yards)]
# sort the passing yards
yards = np.sort(yards)
# create a figure
plt.figure()
# create a bar chart of the players and their passing yards
plt.bar(players, yards)
# rotate the x axis labels
plt.xticks(rotation=90)
# show the plot
plt.show()
```

We gave Copilot this prompt a number of times and it would consistently produce a reasonable graph, but the code and graph varied each time. This is just one representative result, and in this result, the code is notably more advanced than the code you saw earlier this chapter. In fact, we need a more in depth understanding of the code to be able to walk through it in any real detail so we're just going to skip any attempt to read through or interpret the code at this point in the book. We can tell you that, at a high-level, it properly imported a Python module designed to make plots (called matplotlib), did some fairly clever data manipulation in the middle using a Python module called numpy, and even had the sense to rotate player names so that they could print well as an x-axis label.

If you go to run this code, you might hit a snag, however. Because Copilot learned from code in GitHub, it doesn't know what Python modules are installed on your personal machine. The programmers who wrote the code that Copilot learned from likely had matplotlib installed, and matplotlib is the right module to use here, but matplotlib is not a module installed by default in Python. If you don't have it installed, you'll get an error when you try to run the code about not finding the "matplotlib" module.

Python modules

Python modules expand the capability of the programming language. There are many modules in Python and they can help you do anything from data analysis to creating websites to writing video games. You can recognize when code wants to use a Python module by the `import` statement in the code. Python doesn't automatically install all the modules for you because you likely won't use most of them. When you want to use a module then, you'll need to install the package containing the module yourself.

To fix this error, you'll need to install matplotlib. The good news is that Python has made it really easy to install new packages. Go to the "Terminal" at the bottom right of VSCode and type:

```
pip install matplotlib
```

(Note that for some operating systems you may need to use `pip3` rather than `pip`. On Windows machines, we recommend using `pip` if you followed our installation instructions. On Mac or Linux machines, we recommend using `pip3`.)

When you run this command, you'll see that a bunch of modules are installed, including numpy (the next module this code wants to use). (matplotlib requires Python modules of its own, so it installs all the modules you need to use matplotlib in addition to matplotlib itself.) When you try to run the code again, you'll get a plot like this.

Figure 2.5 The plot produced by the code in listing 2.2.
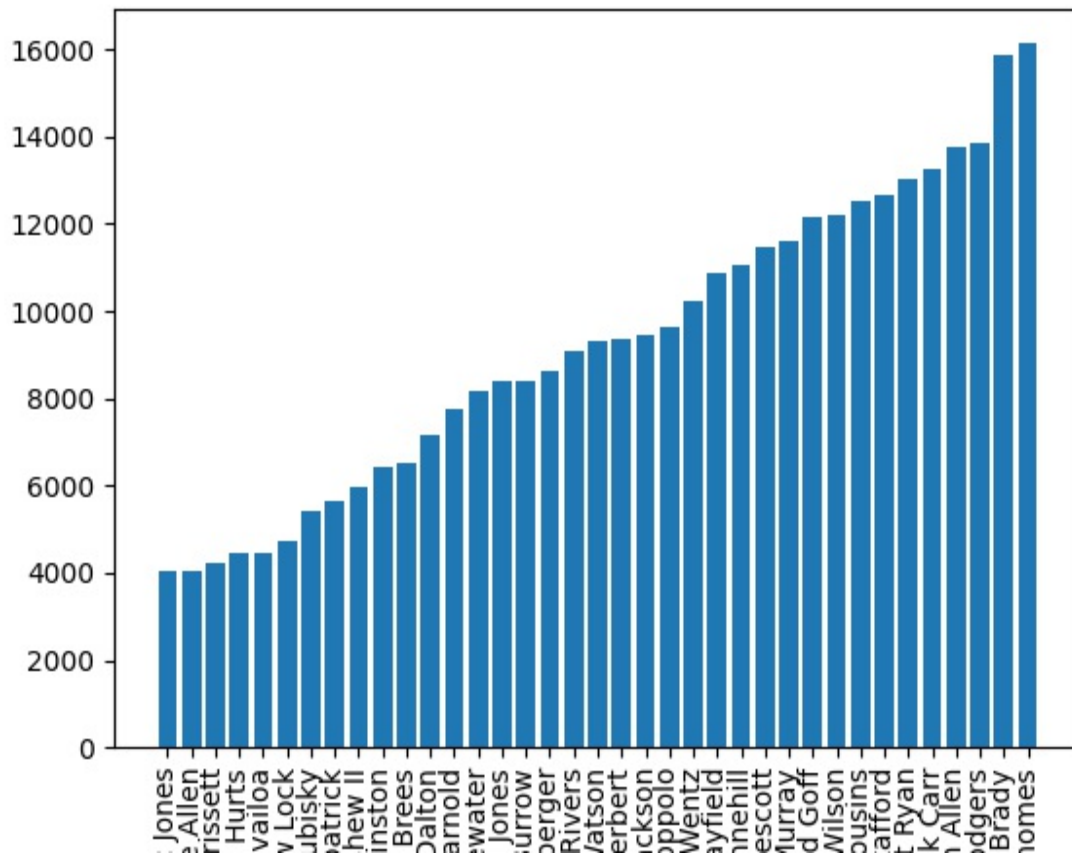
In this bar graph, we see the y-axis is the number of passing yards and the x-axis is the player's name. The players are sorted from fewest yards (with a minimum of 4000) to most yards. Admittedly, it's not perfect as it is missing a y-axis label and the names on the x-axis are cut off at the bottom, but this is pretty impressive given all we gave Copilot was a pretty short prompt. We could keep adding prompts to see if we can format the graph better, but we've already achieved the primary goals for this section which was to show you how powerful Copilot is at helping us write code and to get a feel for how to interact with Copilot.

Indeed, in this chapter, we've accomplished a great deal! **If you've finished setting up your programming environment and followed along the example with us, you should be proud. You've taken a huge step toward writing software!** Beyond the details of setting up your environment, we've written software to solve our first problem. Moreover, you've observed the

process of writing software with Copilot that starts with writing good prompts to help Copilot give us the code we want. In the examples in this chapter, Copilot gave us the code we wanted without us needing to change the prompt or debug the code to figure out why it's not working properly. That was a nice way to showcase the power of using an AI Assistant to program, but you will often find yourself having to test the code, change the prompts, and sometimes try to understand why the code is wrong. This is the AI Assistant programming process that we'll learn more about in upcoming chapters.

2.6 Summary

- You installed Python and VSCode and set up Copilot so you are able to work along with the book and start writing code yourself!
- The VSCode interface has areas for file management, code editing, and running code that will be used throughout the book.
- Prompts are how we tell Copilot to generate code and, when written carefully, can be a highly effective way of creating software.
- Data analysis is a common programming task and .csv files are a common way for storing data to be processed by computers.
- Copilot may generate code that requires you to install additional Python modules.
- Copilot is a powerful tool that is capable of producing code that is as sophisticated (or more) as that produced by college students finishing their first programming course.

2.7 References

[1] C. Alavarado, M. Minnes, L. Porter. "Object Oriented Java Programming: Data Structures and Beyond Specialization." https://www.coursera.org/specializations/java-object-oriented (Accessed Apr. 9, 2023)

[2] S. Valstar, W. G. Griswold, and L. Porter. "Using DevContainers to Standardize Student Development Environments: An Experience Report." In Proc. of 2020 ACM Conference on Innovation and Technology in Computer

Science Education, July 2020, pp. 377–383.

[3] Visual Studio Code. "User Interface."
https://code.visualstudio.com/docs/getstarted/userinterface (Accessed Apr. 9, 2023)

[4] Kaggle. Kaggle Inc. https://www.kaggle.com/ (Accessed Apr. 9, 2023.)

3 Designing Functions

This chapter covers

- Functions in Python and their role in designing software
- Reasonable tasks for Copilot to solve
- The standard workflow of interacting with Copilot
- Examples of writing good functions using Copilot

One of the hardest challenges for programming novices is to know what a reasonable task is to give to Copilot so that it finds a good solution. If you give it too big a task, it will often fail in some fairly spectacular ways and in ways that are extremely difficult to fix. What then, is a reasonable task?

This question is important for our use of Copilot but goes far beyond it. Human programmers struggle with complexity, too. If experienced software engineers try to write code to solve a problem that's too complex without breaking it down into smaller, more solvable subparts, they often have trouble as well. The solution for humans has been to use something called a *function* whose job is to perform one task. There are various rules of thumb for how to write a reasonable function in terms of number of lines of code, but fundamentally these rules try to capture being able to write something that 1) performs a single task and 2) is not so complex that it's hard to get right.

For students who learned to program the old-fashioned way, without Copilot, they'll have been struggling with syntax in code that's 5-10 lines long for about a month before we typically introduce functions and, at that point, it's a natural segue to tell them not to write more code in a single function than they can handle testing and debugging. As you all are learning to work with Copilot rather than syntax directly, our job in this chapter is to teach you about functions and what are reasonable, and unreasonable, tasks to ask Copilot to solve in a single function.

To help you gain perspective on functions, we'll provide you with a number

of examples in this chapter. For those examples, you'll see us employ the central workflow of interacting with Copilot; specifically, the cycle of writing prompts, receiving code from Copilot, and testing to see if that code is correct. In the functions produced by Copilot, you'll begin to see the core programming tools, like loops, conditionals, and lists, that we'll expand on in the next chapter.

3.1 Functions

Before we can learn about the details of writing a function, we need some insight into their purpose in software. Functions are small tasks that help accomplish larger tasks, which in turn help solve larger tasks, and so forth. We think you already have a lot of intuition about breaking apart large tasks into smaller tasks that you'll be able to bring to bear in this example:

Suppose that you've found a word search puzzle in the newspaper that you'd like to solve (See figure 3.1 for an example puzzle). In these kinds of puzzles, you're looking for each of the words in the word list. The words can be found going from left to right, right to left, up to down, or down to up.

Figure 3.1 Example Wordsearch puzzle

```
R  M  E  L  L  L  F  I  L  A  Z  K
B  F  W  H  F  M  O  Z  G  L  Z  C
B  D  T  U  C  N  G  S  L  S  H  A
Y  Y  O  F  U  N  C  T  I  O  N  T
F  A  H  S  I  L  T  A  S  K  O  C
H  N  H  J  O  H  E  L  L  O  C  A
Y  F  M  P  I  P  W  L  B  T  R  J
L  N  S  J  N  E  Z  Y  Z  Z  I  T
```

Find the following hidden words in the puzzle:

CAT   FUNCTION   TASK
DOG   HELLO

At a high level, your task is "find all of the words in the word search".
Unfortunately, that description of the task isn't helpful on its own. It doesn't
tell us what steps we need to carry out to solve the problem.

Try working on the problem right now for a couple minutes. How did you

start? How did you break down the overall task to make it more achievable?

One thing you might do is say, "OK, finding every word is a big task, but a smaller task is just finding the first word (CAT). Let me work on that first!" This is an example of taking a large task and breaking it into smaller tasks. To solve the entire puzzle, then, you could repeat that smaller task for each word that you need to find.

Now, how would we find an individual word, such as CAT? Even this task can be broken down further to make it easier to accomplish. For example, we could break it into four tasks: search for CAT from left to right, search for CAT from right to left, search for CAT from top to bottom, and search for CAT bottom to top. Not only are we making simpler and simpler tasks, but we're also organizing our work into logical pieces. And, most importantly, as we'll see throughout the chapter, it's these simpler tasks whose code we're going to ask Copilot to write and ultimately assemble into our complete programs.

Taking a large problem and dividing it into smaller tasks is called *problem decomposition* and is one of the most important skills in software design. We've dedicated an entire later chapter to it. For now, what is essential is knowing what's too big of a task to ask Copilot. Asking it to make a new video game that's a combination of Wordscapes meets Wordle is not going to work at all. However, you can get Copilot to write a function that's important to solve a larger problem; for example, you might have a function that checks to see if the word provided by the player is present in a list of valid words. Copilot can solve that problem well and that function would help Copilot get closer to being able to solve the larger problem.

3.1.1 The components of a function

The origin of the name "function" goes back to math where functions define the output of something based on an input. For example, if $f(x) = x^2$ we can say that when x is 6, $f(x)$ is 36. As programming functions also have expected output for a particular input, the name is quite apt for programming as well.

As software engineers, we also like to think of functions as promises or

contracts. If there is a function called "larger" and we're told that it takes two numbers and gives us the larger of the two, we have faith that when we give the function the numbers 2 and 5, it will return the answer of 5. We don't need to see how that function works to be able to use it, any more than we need to know how the mechanics of a car works to be able to use the brake pedal. Press the brake, the car slows down. Give the function two numbers, it gives us back the larger of the two.

Every function in Python has a function header (also called a signature), which is the first line of code of the function. Given their ubiquitous nature, we'll want to be able to read and write headers. The function header describes the name of the function and its inputs. In some other languages, it sometimes includes some information about what the output looks like, but in Python you have to find that elsewhere in the code.

In Chapter 2, we wrote # comments to tell Copilot what to do. We can continue to use that approach if we want Copilot to generate a function. For example, we can use comments to ask Copilot to write a function that tells us which of two numbers is biggest:

```
# write a function that returns the larger of two numbers
# input is two numbers
# output is the larger of the two numbers
def larger(num1, num2): #A
    if num1 > num2:
        return num1
    else:
        return num2
```

As with the code in the last chapter, we just wrote the comments to prompt Copilot to give us the code. The function header has three main components: the keyword that tells Python that this is a function, the name of the function, and the inputs to the function. The word `def` denotes that this is creating (defining) a function. After `def` is the name of the function; that name should describe the behavior of the function as well as possible. The name of this function is `larger`. If it's hard to name a function because it does a bunch of different things, that's usually a clue that it's too big a task for a single function, but more on that later. What appears in the parentheses are the parameters. Parameters are how you provide information to a function that it

needs to run. A function can have any number of parameters and some functions have no parameters. This function has two parameters named `num1` and `num2`; there are two parameters because it needs to know the two numbers it's comparing. There can be only one output of a function; the keyword to look for when determining what the function is returning is **return**. Whatever follows the word `return` is the output of the function. In this code, either `num1` or `num2` will be returned. Functions are not required to return anything (e.g., a function that prints a list to the screen has no reason to return anything) so if you don't see a return statement, it isn't necessarily a problem as the function may be doing something else (interacting with the user, for example) rather than returning something. Functions also either have to return something or not return something, they can't return something in some cases and nothing in other cases.

Although we had Copilot generate this function using # comments, this approach is actually a lot of work for Copilot. It first has to get the header right, including figuring out how many parameters you need. Then it has to get the actual code of the function right.

There's an alternate way to prompt Copilot to write the code for a function that may help it generate code more accurately and may help us better understand exactly what we want our function to do. It involves writing a docstring, and we'll use docstrings to write functions for the majority of the book.

Docstrings explain function behavior

Docstrings are how Python functions are described by programmers. They follow the function header and begin and end with three quotation marks.

By writing the header and docstring, you'll make it easier for Copilot to generate the right code. In the header, you will be the one deciding on the name of the function and will provide the names of each parameter that you want the function to use. After the function header, you'll provide a docstring which tells Copilot what the function does. Then, just as before, Copilot will generate the code for the function. Because we gave Copilot the function header, it will be able to learn from the header and is less likely to make mistakes.

Here's what the alternate approach would look like when writing that same `larger` function:

```python
def larger(num1, num2):
    """ #A
    num1 and num2 are two numbers. #A
    #A
    Return the larger of the two numbers. #A
    """ #A
    if num1 > num2:
        return num1
    else:
        return num2
```

Notice that we wrote the function header as well as the docstring; Copilot supplied the body of the function.

3.1.2 Using a function

Once we have a function, how do we use it? Thinking back to our $f(x) = x^2$ analogy, how do we give the function a value of x that is 6 so that it returns to us 36? Let's see how to do this with code by using that `larger` function we just wrote.

The way to use a function is to *call* it. Calling a function means to invoke the function on specific values of parameters. These parameter values are called arguments. Each value in Python has a **type**, and we need to take care to give values of the proper type. For example, that `larger` function is expecting two numbers; it might not work in the expected way if we supply things that aren't numbers. When we call a function, it runs its code and returns its result. We need to capture that result so that we can use it later, otherwise it will be lost. To capture a result, we use a *variable*, which is just a name that refers to a value.

Here, we ask Copilot to call the function, store the result in a variable, and then print the result.

```python
# call the larger function with the values 3 and 5
# store the result in a variable called result
# then print result
result = larger(3, 5) #A
```

```
print(result)
```

The code correctly calls `larger`. Notice that it puts the two values we want compared after the opening parenthesis. When the function finishes, it returns a value that we assign to result. Then we print the result. If you run this program, you'll see that the output 5 gets produced, and that's because 5 is the `larger` of the two values that we asked about.

It's okay if you aren't comfortable with all the details here, but what we want you to be able to recognize is when a function is being called, e.g.,

```
larger(3, 5)
```

The general format for a function call is the following:

```
function_name(argument1, argument2, argument3,... )
```

So, when you see those parentheses right after a name, it means there's a function call. Calling functions as we did here will be important to our workflow with Copilot, particularly in how we test functions to see if they are working properly. We'll also need to call functions to get work done, as functions don't do anything until we call them.

3.2 Benefits of Functions

We already mentioned that functions are critical in performing problem decomposition. Beyond problem decomposition, functions are valuable in software for a number of other reasons, including:

- **Cognitive Load.** You may have heard of cognitive load [1] before. It's the amount of information your brain can handle at any given time and still be effective. If you are given four random words and asked to repeat them back, you might be able to do that. If you are given the same task with 20 words, most of us would fail as it's too much information to handle at once. Similarly, if you've ever been on a road trip with your family and are trying to optimize the travel time, combined with stops for the kids, lunch breaks, bathroom stops, gas station stops, good locations for hotels, and so on, you might have felt your head swimming

to manage all those constraints at once. That point when you can't handle it all at once is when you've exceeded your own brain's processing power. Programmers have the same problem. If they are trying to do too much at once or solve too complex a problem in one piece of code, they struggle to do it correctly. Functions are designed to help programmers avoid doing too much work at once.

- **Avoid Repetition.** Programmers (and, we'd argue, humans in general) aren't very excited about solving the same problem over and over. If I write a function that can correctly compute the area of a circle once, I don't need to write that code ever again. That means if I have two sections of my code that need to compute the area of a circle, I'd write one function that computes the area of the circle and then I'd have my code call that function in each of those two places.
- **Improve Testing.** It's a lot harder to test a section of code that does multiple things compared to code that does one thing. Programmers at companies use a variety of testing techniques, but a key technique is known as *unit testing*. Every function takes some input and produces some output. For a function that computes the area of a circle, for example, the input would be the circle's radius and the output would be its area. Unit tests give a function an input and then compare that input to the desired result. For the area-of-a-circle function, we might test it by giving it varying inputs (e.g., some small positive numbers, some large positive numbers, and 0) and compare the result of the function against the values we know to be correct. If the answers from the function match what we expect, we have a higher degree of confidence that the code is correct. If the code produces a mistake, we won't have much code to check to find and fix the problem. But if a function does more than one task, it vastly complicates the testing process as you need to test each task and the interaction of those tasks.
- **Improve Reliability.** When we write code as experienced software engineers, we know we make mistakes. We also know Copilot makes mistakes. If you imagine you are an amazing programmer and each line of code you write is 95% likely to be correct, how many lines of code do you think you can write before at least one of those lines is likely to be incorrect? The answer ends up being only 14. We think 95% correctness per line is probably a high bar for even experienced programmers and is likely a higher bar than what Copilot produces. By keeping the tasks

small, things solvable in 12-20 lines of code, we reduce the likelihood that there's an error in the code. If combined with good testing as noted above, we can feel even more confident that the code is correct. Lastly, nothing is worse than code that has multiple mistakes that interact together, and the likelihood of multiple mistakes grows the more code you write. Both of us have been on multi-hour debugging expeditions because our code had more than one mistake and, for both of us, we became a lot better at frequent testing of short pieces of code as a result!

- **Improve Code Readability**. In this book, we're going to mostly use Copilot to write code from scratch. But that's not the only way to use Copilot. If you have a larger piece of software that you or your coworkers are all editing and using, Copilot can jump in to help write code for that, too. It's in everyone's interest to understand the code, whether most of it is written by humans or by Copilot. That way, we can find bugs more easily, determine what code to start modifying when we want to add new features, and understand at a high level what would be easy or difficult to achieve with our overall program design. Having tasks broken down into functions helps us understand what each part of the code is doing so we can gain a better understanding of how it all works together. It also helps divide up the work and responsibility for ensuring the code is correct.

These benefits are huge for programmers. Programming languages haven't always had functions. But even before they did, programmers did their best to use other features to mimic functions. They were ugly hacks (Google "goto statements" if you're interested), and all programmers are happy that we have proper functions now.

You may be asking, "I see how these advantages matter to humans, but how do they impact Copilot?" In general, we believe all the principles that apply to humans apply to Copilot, albeit for sometimes different reasons. Copilot may not have cognitive load, but it's going to do better when we ask it to solve problems similar to what's been done by humans before; since humans write functions to solve tasks, Copilot will mimic that and write functions as well. Once we've written and tested a function, whether by hand or by Copilot, we don't want to write it again. Knowing how to test if your program is working properly is just as essential for code produced by humans

as it is by Copilot. Copilot is as likely to make mistakes when it generates code, so we want to catch them quickly, just as we do with human-written code. And even if you only work on your own code and never have anyone else read it, as programmers who have had to go back to edit code we wrote years ago, let us tell you that it is important for your code to be readable, even if the only person reading it is you.

3.3 Roles of functions

Functions are used in many different roles in programming. At a high level, programs are really just functions that (often) call other functions. Critically, all programs, including Python programs, originate with a single function (named `main` in languages like Java, C, and C++). Python's `main` function is essentially the first line of code that isn't in a function. But if every program starts with a single function and we just told you that trying to solve a large problem with a single function is a mistake, how does that work? Well, `main` will call other functions which, in turn, will call other functions, and so forth. The code will still execute (mostly) sequentially in each function, so it may start in `main`, but then move to another function, and so forth.

As an example, let's use the following code. We wrote this code, not Copilot, because no one would ever want to write this code for anything useful outside teaching. It's just for demonstrating how function calls work.

Listing 3.1 Python code to demonstrate how Python handles function calls

```python
def funct1():
    print("there")
    funct2()
    print("friend")
    funct3()
    print("")

def funct2():
    print("my")

def funct3():
    print(".")

def funct4():
```

```
    print("well")

print("Hi") #A
funct1()
print("I'm")
funct4()
funct3()
print("")
print("Bye.")
```

If we were to run this program, this would be the output (we'll explain why below):

```
Hi
there
my
friend
.

I'm
well
.

Bye.
```

In figure 3.2, we are providing a diagram of how the code in listing 3.1 would be executed by the computer. We've intentionally provided an example that has many function calls to tie together what we just learned. Again, this is NOT practical code; it's just for learning purposes. Let's trace through the code execution together. It may be easier to refer to figure 3.2 than listing 3.1 as you follow along, but either will work.

The program will start execution with the first line in the Python code that isn't a function (`print("Hi")`). Although Python doesn't have a `main` function per-se, we'll refer to the block of code after the functions as `main` to help this explanation. Code executes sequentially unless it encounters commands that tell it to execute code somewhere else. So, after executing `print("Hi")`, it will go to the next line which is the call to funct1: `funct1()`. The call to funct1 changes where the code is executing to the very start of that function which is the statement: `print("there")`. The next line of funct1 calls funct2, so this means the program will execute the first line of funct2: `print("my")`. What gets really interesting is what happens when funct2
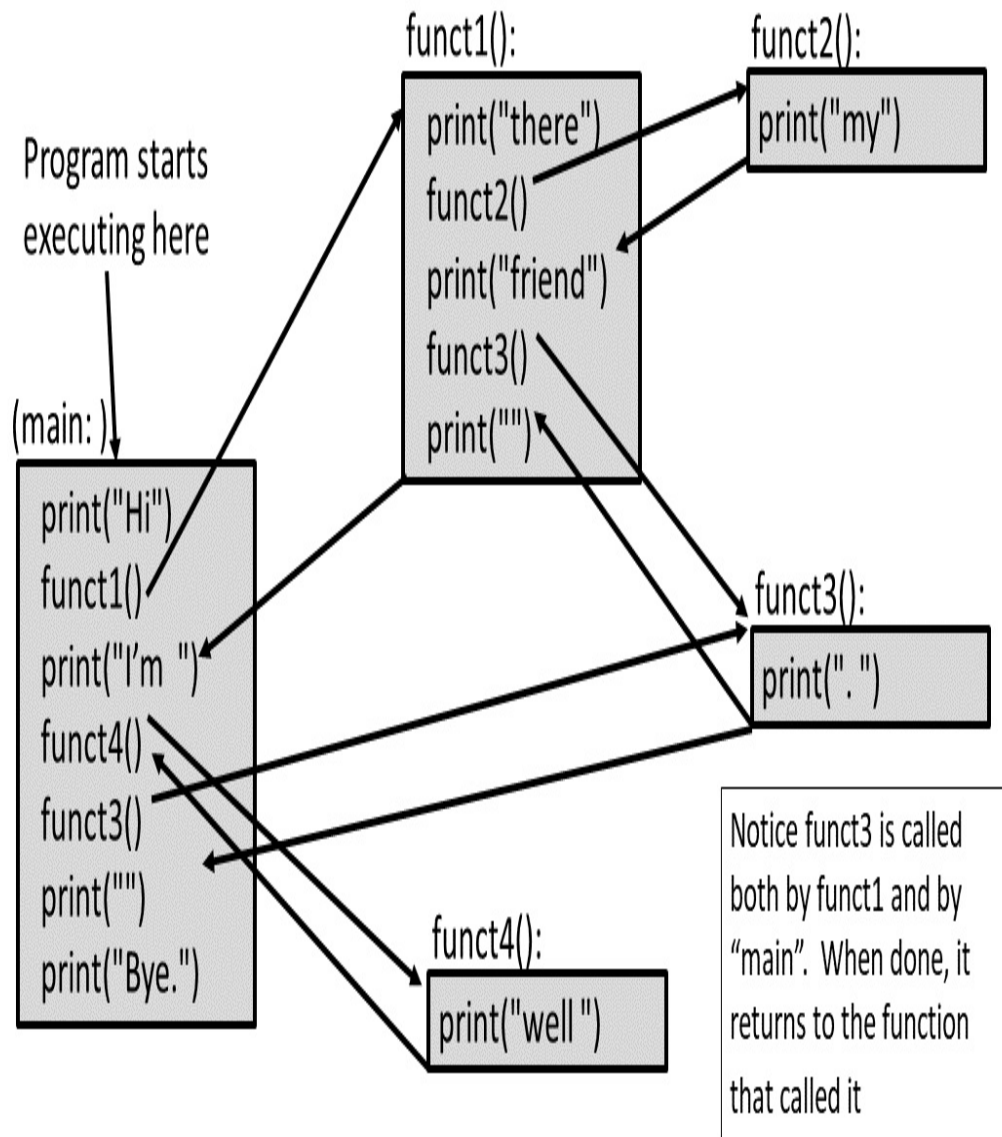
finishes. There are no more lines of code to execute, so it automatically moves execution back to the first line following the call to funct2 in funct1. (If the function call is in the middle of another statement, that statement resumes execution, but for this example, the function calls are each on their own line.) If you are curious why it goes to the next line after the call to funct2 rather than back to the call of funct2, the issue is if it returned back to the call to funct2, it would be trapped calling funct2 forever. As a result, functions always return back to the next piece of code to execute (in this example, the next line) after they were called.

Continuing this example, the next line of the code executed will be the line that prints "friend". The next line calls funct3. funct3 prints a "." and then returns back to its caller.

So, we're back in funct1, on the line `print("")`. Printing an empty piece of text just causes a new line. Now funct1 is finished, so it transfers execution back to the next line in main after it was called. We suspect you are getting the idea by now, so let's move a bit more quickly:

- Main next prints "I'm" and then calls funct4.
- funct4 prints "well" and then returns back to main where the next line of code calls funct3.
- funct3 prints a "." and then returns back to main. Notice that funct3 was called both by funct1 and by main, but that's okay because functions remember how to return back to the function that called them. In fact, having multiple functions calling the same function is a sign that the function being called multiple times is a good function, because of its reuse.
- After funct3 returns to main, it will print a "" which causes a new line to be started, and then it prints the word "Bye."

Figure 3.2 Flow of function execution in our example from listing 3.1.

funct1():

```
print("there")
funct2()
print("friend")
funct3()
print("")
```

funct2():

```
print("my")
```

Program starts
executing here

(main:)

```
print("Hi")
funct1()
print("I'm ")
funct4()
funct3()
print("")
print("Bye.")
```

funct3():

```
print(". ")
```

funct4():

```
print("well ")
```

Notice funct3 is called
both by funct1 and by
"main". When done, it
returns to the function
that called it

That was a long example, but we provided it to help give you an idea of how functions execute and how programs consist of defining and calling functions. In any software you use, think about the specific tasks that it performs: the programmers probably wrote one or more functions for each one. The button in a text editor that changes the text to bold probably calls a

function to change the text to bold. That function might change the editor's internal idea of the text (the editor likely stores your text in a different format than how you view it) and then it might call another function that updates the user's (your) view of the text.

We'd also like to use this example to discuss the different roles that functions play. A *helper* function is a function whose job is to make another functions job easier. In a sense, every function that isn't "main" is a helper function.

Some functions simply call a bunch of other functions without doing any of their own work. There aren't any of these in our example. However, if you removed the three print statements from funct1, it would become this type of coordinating function. Others may call helper function(s), then do some work on their own. funct1 is a great example of a function that calls other functions but also does work on its own. And another group of functions stand on their own, without calling other functions (except perhaps functions that already come with Python) for help—we'll call these functions *leaf* functions. Why leaf? If you imagine all of the function calls as a big tree, these functions are the leaves of the tree as they have nothing coming out of them. funct2, funct3, and funct4 are all leaf functions in our example. We're primarily concerned with leaf functions in this chapter, but you'll see examples of other kinds of functions here and especially in later chapters.

3.4 What's a reasonable task for a function?

There's no clear rule for what makes a good function, but there are some intuitions and recommendations we can share. Make no mistake, though: identifying good functions is a skill that takes time and practice. To help you with this, in this section we'll outline our recommendations and provide you with some good and bad examples to help build that intuition. Then, in Section 3.5, we'll show you a bunch of examples of how to write good functions.

3.4.1 Attributes of good functions

Here are some guidelines that we believe will help you see what makes a good function:

- **One clear task to perform.** If they are leaf functions, this might be something like, "compute the volume of a sphere", "find the largest number in a list", or "check to see if a list contains a specific value." Non-leaf functions can achieve more broad goals, like "update the game graphics" or "collect and sanitize input from the user". Non-leaf functions should still have a particular goal in mind but are designed knowing they will likely call other functions to achieve their goal.
- **Clearly defined behavior.** The task "find the largest number in a list" is clearly defined. If I gave you a list of numbers and asked you for the largest number, you know what you should do. In contrast, the task "find the best word in the list" is poorly defined as stated. You need more information – what's the "best" word? Is it the longest, the one that uses the fewest vowels, the one that doesn't share any of the same letters as "Leo" or "Dan"? You get the point; subjective tasks aren't great for computers. Instead, we could write the function "find the word in the list that has the most characters" as what is expected is well defined. Often programmers can't put all the particulars of a function just in the name, so they fill in the details in the docstring to clarify its use. If you find yourself having to write more than a few sentences to describe the functions behavior, the task is probably too much for a single function.
- **Short in number of lines of code.** We've heard different rules over the years for the length of functions, informed by different company style guidelines. The lengths we've heard vary from 12 lines to 20 lines of Python code as the maximum number of lines. In these rules, the number of lines is being used as a proxy for code complexity and it's not a bad general rule of thumb. As programmers ourselves, we both apply similar rules to our code to ensure the complexity doesn't get out of hand. With Copilot, we can use this as a guide as well. If you ask Copilot for a function and it gives you back 50 lines of code, this probably isn't a good function name or task and, as we discussed earlier, that many lines of code are likely to have errors anyway.
- **General value over specific use.** A function that returns the number of values in a list that are greater than 1 might be a specific need for a part of your program, but there's a way to make this better. The function should be rewritten to return the number of values in the list that are greater than another parameter. The new function would work for your use case (give the function 1 for the 2nd parameter) and for any other

value than 1. We strive to have functions be as simple, but as powerful as possible.

- **Clear input and output.** You generally don't want a lot of parameters. That doesn't mean you can't have a lot of input though. A single parameter could be a list of items (we'll talk more about lists soon). It does mean that you want to find ways to keep the number of inputs to a minimum. You can only return one thing, but again, you can return a list so you aren't as limited as it may appear. But if you find yourself writing a function that sometimes returns a list, sometimes returns a single value, and sometimes returns nothing, that's probably not a good function.

3.4.2 Examples of good (and bad) leaf functions

Here are examples of good leaf functions:

- *Compute the volume of a sphere—given the sphere's radius, return its volume*
- *Find the largest number in a list—given a list, return the largest value*
- *Check to see if a list contains a specific value—given a list and a value, return true if the list contains the value and false if it does not*
- *Print the state of the checkers game—given a 2D list representing the game board, output the game board to the screen in text*
- *Insert a value in a list—given a list, a new value, and a location in the list, return a new list that is the old list with the new value inserted at the desired location*

Here are examples of bad leaf functions and our reasons for why they are bad:

- *Request a user's tax information and return the amount they owe this year.* Perhaps in some countries this wouldn't be too bad, but we can't imagine this as a single function in either the United States or Canada given the complexity of the tax rules!
- *Identify the largest value in the list and remove that value from the list.* This might not seem so bad, but it's really doing two things. The first is to find the largest value in the list. The second is to remove a value from

the list. We'd recommend two leaf functions, one that finds the largest and one that removes the value from the list. However, this might make a good non-leaf function if your program needs to perform this task frequently.

- (Thinking of our dataset from Chapter 2) *Return the names of the quarterbacks with more than 4000 yards of passing in the dataset.* This has too much specificity. Without a doubt, the number 4000 should be a parameter. But it'd likely be better to make a function that takes as input the position (quarterback, running back), the statistic (passing yards, games played), and the cutoff that we care about (4000, 8000), all as parameters. This new function would provide far more capability than the original, allowing a user to call the function to determine not only the names of particular quarterbacks who threw for more than 4000 yards, but also running backs who had more than 12 rushing touchdowns.
- *Determine the best movie of all time.* This is too vague. Best movie by what definition? What movies should it consider? A better version of this might be a function that determines the highest rated movie by users given at least a minimum number of ratings. This function would likely be part of a larger program where the function would have, as input, data from a movie database (say IMDB) and minimum number of user ratings. The output of the function would be the highest rated movie that has at least as many ratings as specified.
- *Play Call of Duty.* This might be the "main" function in the large code base for the game Call of Duty but it is definitely not a leaf function.

3.5 The cycle of design of functions with Copilot

Designing functions with Copilot involves the following cycle of steps (see figure 3.3):

1. Determine the desired behavior of the function
2. Write a prompt that describes the function as clearly as possible.
3. Allow Copilot to generate the code.
4. Read through the code to see if it seems reasonable.
5. Test the code to see if it is correct.
   a. If the code is correct after multiple tests, move on.

b. If the code is incorrect, move to step 2 and edit the prompt.

Figure 3.3 General editing cycle with Copilot. This assumes you define a reasonable function.

Define the desired behavior of the function	→	Write a prompt that describes the function	→	Allow Copilot to generate the code	→	Examine the code to see if it is reasonable

Test the code to see if it is correct

incorrect

correct

Done
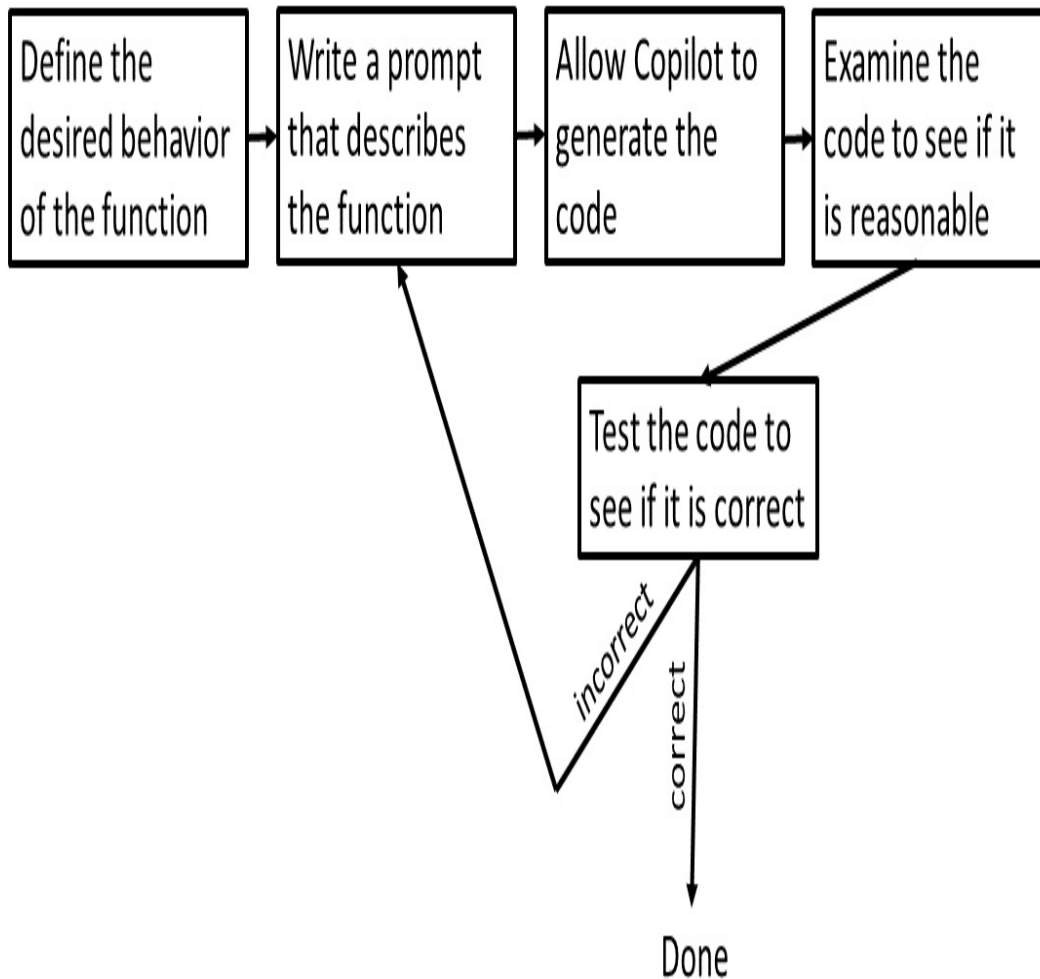
We won't learn how to do step 4 until the next chapter, but we bet you can

already recognize when the code is blatantly wrong. For example, Copilot might give you only comments to fill the body of the function. Comments don't do anything — they are not code! — so a bunch of comments with no other code is clearly not the right thing to do. Or it might just write a single line "return -1". Or, our personal favorite, "Your code here". Copilot learned that one from us professors when we provide students partial code and ask them to write the rest with "Your code here". Those are all obviously incorrect, but in the next chapter, we'll go over how to read code so you can more quickly spot when more complicated code is incorrect and, perhaps more importantly, see where and how to fix it. In later chapters, we'll keep expanding on this cycle to include effective debugging practices and we'll keep practicing how to improve prompts.

3.6 Examples of creating good functions with Copilot

In this section, we're going to write a bunch of functions with Copilot. We'll be coding them entirely in Copilot to help you see the cycle of function design we just described. Although our goal in this chapter isn't to help you read code just yet, we will be seeing programming features (sometimes called constructs) in the solutions that are very common in code (e.g., if statements, loops) so we'll point those out when we see them. Then in Chapter 4, we'll say more about how to read this code in more detail.

Many of the functions we're about to work on are unrelated to each other. For example, we'll start with a function about stock share prices and move to functions about strong passwords. You typically wouldn't store unrelated stuff like this in the same Python file. But as we're just exploring different examples of good functions, please feel free to store all functions in the same Python file, perhaps named function_practice.py or ch3.py.

3.6.1 Dan's stock pick

Dan is an investor in a stock called AAAPL. He purchased 10 shares for $15 each. Now, each of those shares is worth $17. Dan would like to know how much money he has made on the stock.

Remember that we want to make our function as general as possible. If the only thing our function could do is calculate this exact AAAPL situation, it wouldn't be that useful in general. Sure, it would help Dan right now, but what about when AAAPL's share price changes again, or when he is interested in another stock entirely?

A useful general function here would take three parameters, all of which are numbers. The first parameter is the number of shares purchased, the second is the share price when the shares were purchased, and the third is the current share price. Let's call this function money_made, since it's going to determine how much money we've made or lost on the stock. In general, you want to name your function as an action word or words that describe what your function is doing. With that, we have enough to write the function header:

```
def money_made(num_shares, purchase_share_price, current_share_pr
```

Now we need a docstring. In the docstring, we need to explain what each parameter is for by using its name in a sentence. We also need to include what the function is supposed to do.

Adding our docstring, here is the full prompt that we provide to Copilot:

```
def money_made(num_shares, purchase_share_price, current_share_pr
    """
    num_shares is the number of shares of a stock that we purchas
    purchase_share_price is the price of each of those shares.
    current_share_price is the current share price.

    Return the amount of money we have earned on the stock.
    """
```

After typing that prompt, go to the next line and press the tab key. Copilot will fill in the code for the function. Don't worry that the code gets indented: the code of functions is supposed to be indented, and in fact it's an error if it isn't!

Here's what we got from Copilot:

```
return num_shares * (current_share_price - purchase_share_price)
```

This code seems sensible. In the parentheses it figures out the difference between the current price and the purchase price (the - is used for subtraction), and then it multiplies that by the number of shares that we have (the * is used for multiplication). Inspecting code like this is a useful skill, and we'll get serious about it in the next chapter. Another useful skill is testing the function.

To test the function, we call it using various inputs and observe the output in each case. We could do this by asking Copilot to call the function and then running our program, much as we did with our "larger" function. We could then ask Copilot to change the function call by asking it to call the function with a different input, and run our program again, repeating as many times as needed.

However, we find it easier and more convenient to call the function ourselves from an interactive window.

This way, we can call the function as many times as we like without going through Copilot at all, and without cluttering up our program with stuff we're going to delete anyway. To try this interactive approach, select/highlight all of the code of the function, then press <Shift>+Enter (you can access a similar interactive session by selecting the text, right-clicking, and choosing Run Selection/Line in Python Window, but the guidance here is if you use <Shift>+Enter). Figure 3.4 shows what this looks like if you select the text of the function and press <Shift>+Enter.

Figure 3.4 Running Python in an interactive session in the Terminal of VSCode. Note the >>> at the bottom of the Terminal.
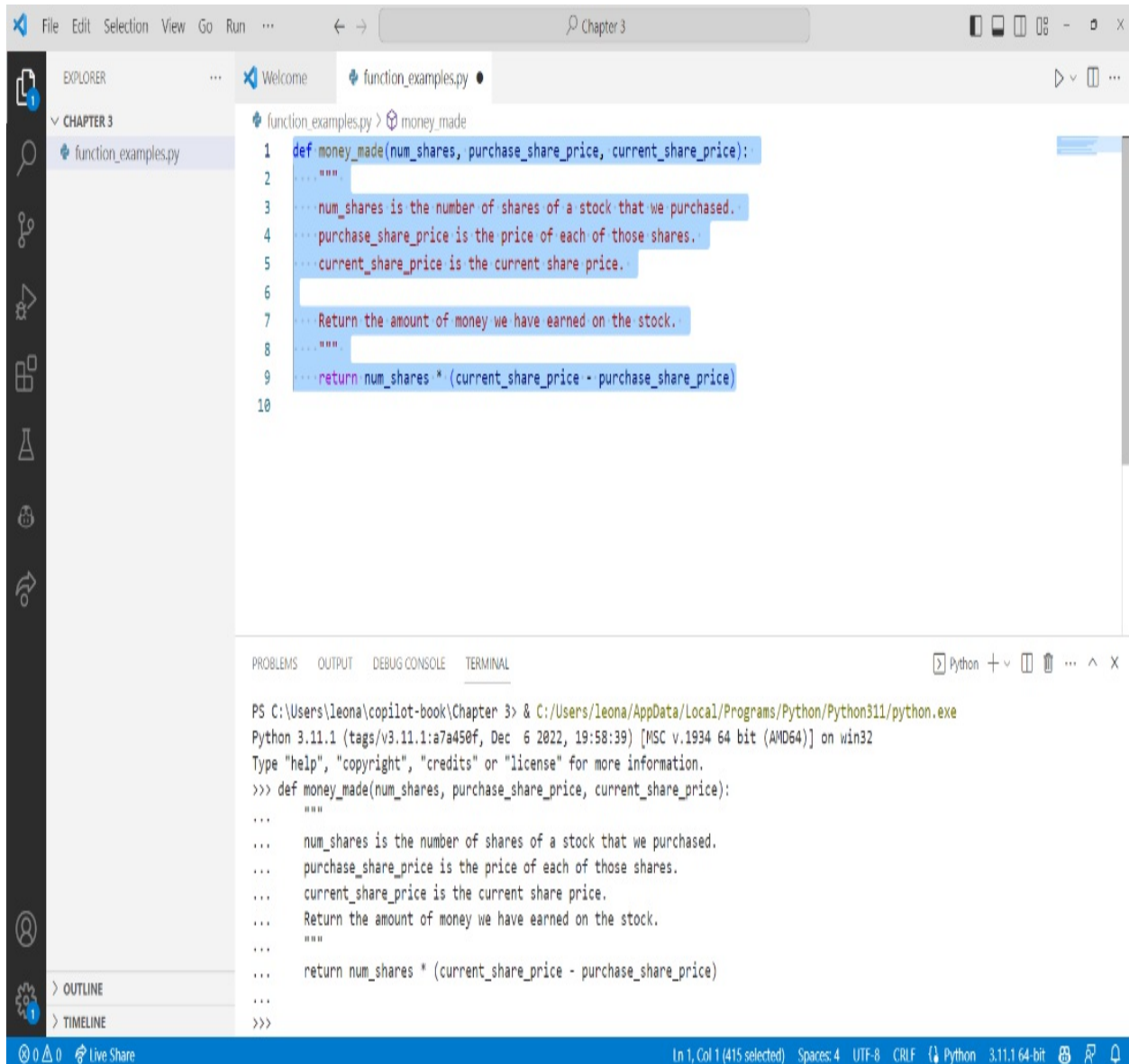
At the bottom of the resulting window, you will see three greater-than symbols >>>. This is called a *prompt*, and you're allowed to type Python code here. (This *prompt* has nothing to do with the kind of prompt that we use when interacting with Copilot.) It will show us the result of the code that we type right away, which is convenient and fast.

To call our `money_made` function, we need to provide three arguments, and they will be assigned left to right to the parameters. Whatever we put first will be assigned to `num_shares`, whatever we put second will be assigned to `purchase_share_price`, and whatever we put third will be assigned to `current_share_price`.

Let's try this! At the prompt, type the following and press enter (or <shift>+Enter). Don't type the >>> as that's already there, and we are including it throughout the book to make it clear where we are typing. Please see figure 3.5 for an example of running the function in the terminal at the Python prompt.

```
>>> money_made(10, 15, 17)
```

You'll see an output of

```
20
```

Is 20 correct? Well, we bought 10 shares, and each of them went up $2 (from $15 to $17), so we did make $20. Looks good!

Figure 3.5 Calling the `money_made` function from Python prompt in the VSCode Terminal.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Type "help", "copyright", "credits" or "license" for more information.
>>> def money_made(num_shares, purchase_share_price, current_share_price):
...     """
...     num_shares is the number of shares of a stock that we purchased.
...     purchase_share_price is the price of each of those shares.
...     current_share_price is the current share price.
...     Return the amount of money we have earned on the stock.
...     """
...
...     return num_shares * (current_share_price - purchase_share_price)
...
>>> money_made(10, 15, 17)
20
>>>
```

We're not done testing, though. When testing a function, you want to test it in various ways, not just once. All one test case tells you is that it happened to work with the particular input values that you provided. The more test cases

we try, each testing the function in a different way, the more confident we are that our function is correct.

How do we test this function in a different way? We're looking for inputs that are somehow a different *category* of input. One not-so-good test right now would be to say, "what if our stock went from $15 to $18, instead of $15 to $17?". This is pretty much the same test as before, and chances are that it will work just fine.

A good idea is to test what happens when the stock actually *loses money*. We expect to get a negative return value in this case. And it appears that our function works just fine with this category of test. Here's our function call and the output that is returned to us:

```
>>> money_made(10, 17, 15)
-20
```

What other tests can we do? Well, sometimes a stock price doesn't change at all. We expect 0 in this case. Let's verify it:

```
>>> money_made(10, 15, 15)
0
```

Looks good!

Testing is a combination of science and art. How many categories of things are there to test? Are these two calls really two different categories? Have we missed any categories? You will improve your testing ability through practice, and we'll spend all of Chapter 6 talking about testing. For now, it looks like our money_made function is doing its job.

It's possible for a function to use variables (rather than just its parameters) in its code, and we want to show you an example of that now so that you're ready when you see Copilot doing it.

Here's an equivalent way to write the code for our money made function:

```
price_difference = current_share_price - purchase_share_price
    return num_shares * price_difference
```

This may even be easier to read for you: first it figures out the difference in share price, and then it multiplies that by the number of shares. We encourage you to test this version to help convince yourself that it is still correct.

3.6.2 Leo's password

Leo is signing up for a new social network website called ProgrammerBook. He wants to make sure that his password is strong.

Leo starts with a modest definition of what it means for a password to be strong: it's strong if it's not the word 'password' and not the word 'qwerty.' (Those are terrible passwords, for sure, but in reality, we have to do way better than this definition to ensure that our password is strong!)

A helpful function would be one that takes a proposed password and tells us whether it is strong or not.

Unlike our previous functions in this chapter, we're not dealing with numbers here. The parameter, the password to check, is text. And the return value is supposed to indicate some yes/no result. We need new types!

The Python type for text is called a **string**. There are zillions of possible strings, because we can use a string to store whatever text we want. And the Python type for a yes/no result is called a **Boolean** or **bool**. A bool has only two values: True or False.

Alright! We're ready to prompt Copilot. For functions that return bool (True/False) values, we usually name the function as has_x, or is_x, or using some other verb that implies a true/false result.

```
def is_strong_password(password):
    """
    A strong password is not the word 'password'
    and is not the word 'qwerty'.

    Return True if the password is a strong password, False if no
    """
```

In response, here's what Copilot wrote for us:

```
return password != 'password' and password != 'qwerty'
```

Look over that code for a moment. That `!=` operator is the 'not equal' operator, and each string that doesn't count as a strong password is in single quotation marks. We're going to use quotation marks every time we work with a string value. Again, we'll talk more about how to read code in the next chapter, but for now we hope that the code makes a fuzzy sort of sense: it looks like it's checking that the password isn't 'password' and isn't 'qwerty', which matches our definition of a strong password.

Now let's make sure that this function does indeed work, by making a few test calls.

We'll start by testing the passwords that are not strong. We had better get back False for each of these—and we do!

```
>>> is_strong_password('password')
False
```

```
>>> is_strong_password('qwerty')
False
```

Good! Now, let's try a strong password. We need to see True here:

```
>>> is_strong_password('N3w Y0rk J375')
True
```

Good stuff! We now see that the function sometimes returns False and other times returns True, so at least it's not always returning False.

Let's try one more:

```
>>> is_strong_password('Leo')
True
```

Well, technically True is correct, but wow is 'Leo' a bad password. We should really have done better with our definition of a strong password.

Let's change our prompt to give a more reasonable definition of what it means for a password to be strong.

Here's our next prompt and what Copilot gave us.

```python
def is_strong_password(password):
    """
    A strong password has at least one uppercase character,
    at least one number, and at least one special symbol.

    Return True if the password is a strong password, False if no
    """
    return any(char.isupper() for char in password) and \
            any(char.isdigit() for char in password) and \
            any(not char.isalnum() for char in password)
```

That `for char in password` that shows up a few times here is a way for Python to consider each character of the password. It enables something called a loop, which is a key topic that we'll study in Chapter 5.

It's of course possible that you didn't get the same code from Copilot as we did. But based on the code that we got, it looks like the first piece is looking for any uppercase character, the second piece is looking for any digit, and the third is looking for something that's not an `isalnum`.

Maybe `isalnum` means 'is alphanumeric'. Does that include special symbols and, if so, which ones?

Let's test this function to see how well we're doing.

```
>>> is_strong_password('Leo')
False

>>> is_strong_password('N3w Y0rk J37$')
True

>>> is_strong_password('N3w Y0rk J375')
True
```

Uh oh—we got the wrong answer for that third password. It looks pretty strong, but it's not, because it lacks a special symbol. Our function returned True, but that's the wrong return value: it was supposed to return False.

The code that Copilot gave us is wrong. Maybe the code that you got worked fine, but we didn't get so lucky and it's important we know how to identify

(like we just did) when the code is wrong and then fix it. It's not important to know why it's wrong right now, but if you are curious about why our prompt may have given us the wrong behavior, we didn't specify what we meant by a special character and the code it gave us counts spaces as special characters.

At this point, with our wrong code, we have two options. One option is to hit <Ctrl>+Enter and look through the other suggestions that Copilot gives us. Maybe one of those suggestions will have something to do with special symbols that looks different from the one that didn't work?

Indeed, on our computer here, we see such options, including this one:

```
num_upper = 0
    num_num = 0
    num_special = 0
    for char in password:
        if char.isupper():
            num_upper += 1
        elif char.isnumeric():
            num_num += 1
        elif char in '!@#$%^&*': #A
            num_special += 1
    if num_upper >= 1 and num_num >= 1 and num_special >= 1:
        return True
    else:
        return False
```

Test this function with strings like 'Leo' and ''N3w Y0rk J375' and 'N3w Y0rk J375$' and you should see that the function works well.

Our second option is to do some prompt engineering, which means adjusting our prompt to influence what Copilot gives us.

In our original prompt, we talked about "special symbols". This, in retrospect, is pretty vague. We probably meant something more specific, like "punctuation". If we specifically talk about punctuation in our prompt, we get this interaction with Copilot:

```
def is_strong_password(password):
    """
    A strong password has at least one uppercase character,
    at least one number, and at least one punctuation.
```

```
    Return True if the password is a strong password, False if no
    """
    return any(char.isupper() for char in password) and \
           any(char.isdigit() for char in password) and \
           any(char in string.punctuation for char in password)
```

Looks good... that last line is talking about 'punctuation', which is hopeful. Let's test:

```
>>> is_strong_password('Leo')
False

>>> is_strong_password('N3w Y0rk J375')
 Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ch2.py", line 44, in is_strong_password
    any(char in string.punctuation for char in password)
  File "ch2.py", line 44, in <genexpr>
    any(char in string.punctuation for char in password)
                 ^^^^^^
NameError: name 'string' is not defined
```

Look at the bottom of that error message, 'string' is not defined, eh? We're running into an issue that's similar to what we saw in Chapter 2 with modules. Copilot wants to use a module, called string, but it is a module that needs to be imported before we can use it. There are a lot of modules in Python, but the string module is pretty well known. As you work with Copilot more, you'll learn which modules are commonly used so you know to import them. You could also do a quick internet search to ask, "is string a Python module" and the results would confirm that it is. What we need to do is import the module.

Note that this is different, somewhat, from what we encountered in Chapter 2. In Chapter 2, we saw what happens when the code from Copilot imports modules we didn't have installed and we had to install the package containing those modules to fix the issue. In this case, the code from Copilot is using a module that already happens to be installed with Python, but it forgot to import it. So, we don't need to install string; we just have to import it.

Importing Modules

There are a number of useful modules available in Python. We saw how powerful matplotlib is in Chapter 2. But in order for Python code to take advantage of modules, we have to import that module. You might ask why don't we just have modules available to us without importing them, but that would massively increase the complexity of the code and what Python has to do to run code behind the scenes. Instead, the model is to include modules if you want to use them, and they aren't included by default.

Let's 'import string' at the top of our code:

```
import string

def is_strong_password(password):
    """
    A strong password has at least one uppercase character,
    at least one number, and at least one punctuation.

    Return True if the password is a strong password, False if no
    """
    return any(char.isupper() for char in password) and \
           any(char.isdigit() for char in password) and \
           any(char in string.punctuation for char in password)
```

Now we're in good shape:

```
>>> is_strong_password('Leo')
False

>>> is_strong_password('N3w Y0rk J375')
False

>>> is_strong_password('N3w Y0rk J375$')
True
```

That last one is True—it's a strong password!—because it has the $ punctuation added to it.

We hope that you're now convinced of the value of testing! Sometimes our students don't test their code. They assume that the code they write is correct because it made sense to them. An interesting difference between novice and experienced programmers is that novices often assume their code is right, whereas experience programmers assume the code is wrong until thoroughly

tested and proved otherwise. Beyond this, we find that our students sometimes fail to test well because it's disheartening to learn that the code is wrong. But it's better to know now, rather than later when others are using your code in a serious application. Finding errors through testing is actually a good thing.

3.6.3 Getting a strong password

Now that we have a function that tells us whether a password is strong or not, let's write a function that obtains a strong password from the user. It will ask again and again for a password until the user types a strong one. This is the kind of code that websites use when they tell you, "Sorry, your password is too weak, try again."

What should the header for such a function look like? Well, it's going to ask the user for a password, so we won't have a password parameter. In fact, we won't have any parameters at all, because the function doesn't need any information to do its job – it just has to prompt the user and the user is going to type the password at the keyboard. When the function has done its work, it will return the strong password, so it will continue to return a string as the output.

Let's try this prompt:

```
def get_strong_password():
    """
    Keep asking the user for a password until it is a strong
    password, and return that strong password.
    """
```

We got the following code from Copilot as a result:

```
password = input("Enter a strong password: ")
    while not is_strong_password(password): #A
        password = input("Enter a strong password: ")
    return password
```

That 'while' keyword creates another kind of loop, this one continuing as long as the entered password is not strong. Copilot is also smart enough to call our earlier is_strong_password function to determine what counts as a

strong password. As we will see in future chapters, using functions as building blocks in this way is precisely how large programs are built. You will often notice Copilot calling your earlier functions to solve later ones, much as we observed here.

Let's test this! Highlight all of our password function code and hit <Shift>+Enter. We'll call the function that we want to test. Then, try typing passwords, hitting enter after each one. You'll notice that it keeps asking you until you finally provide a strong password:

```
>>> get_strong_password()
Enter a strong password: Leo
Enter a strong password: N3w Y0rk J375
Enter a strong password: N3w Y0rk J375$
'N3w Y0rk J375$'
```

Notice that it stops asking us for a password when we finally provide a strong password. Then, we see the string in quotes that it returned, which is indeed our strong password.

3.6.4 Scrabble scoring

One of Dan's favorite board games is Scrabble. Have you played it? If not, all you need to know is that you have some tiles in your hand, each with a letter on it, and your goal is to form a word using any combination of those letters. You don't need to form the word exclusively with your tiles – you can attach those letters to existing letters on the board to create longer words – but we're not going to worry about that here. The important thing for us is that different letters are worth different numbers of points. For example, an 'a' is worth only one point, because 'a' is such a common letter. But 'q'? 'z'? Those doozies are each worth ten points, because they're so tough to use. … or should we say 'puzzling' to use. Yeah, that's better.

To calculate the score for a word, we add up the scores for each of its letters. For example, the score for 'zap' would be 14. That's because z is worth 10, a is worth 1, and p is worth 3.

Dan would like a function that, given a word, tells him how many points that word is worth. OK, so we need a function that takes a word (which is just one

parameter). That's just one parameter. Let's try this prompt, where we've included the number of points that each letter is worth per official Scrabble rules:

```python
def num_points(word):
    """
    Each letter is worth the following points:
    a, e, i, o, u, l, n, s, t, r: 1 point
    d, g: 2 points
    b, c, m, p: 3 points
    f, h, v, w, y: 4 points
    k: 5 points
    j, x: 8 points
    q, z: 10 points

    word is a word consisting of lowercase characters.
    Return the sum of points for each letter in word.
    """
```

For this function, Copilot isn't filling in the whole body for us with one press of the tab key. Tab is also used to indent the code, which we also need to do here before Copilot will present us with code. So, we're having to press tab (sometimes once for the indentation, then again for Copilot), wait for the code to appear, then press enter to accept that code, then press tab again for more code, and repeat the process.

Getting Copilot to suggest code may require pressing <Tab> or <Enter>

We find that Copilot will sometimes pause in giving suggestions until you press a key to help prompt it. As mentioned in table 2.1, if you are giving it comments, it will sometimes just want to give you more comments as suggestions. Pressing Enter another time can often trigger it to start generating code instead. Other times, pressing tab will cause it to continue providing suggestions. Please refer back to table 2.1 if Copilot is not generating code for you.

Finally, after doing that a few times, the entire code unfurls, and we get the following:

```python
points = 0
    for char in word:
```

```
            if char in "aeioulnstr":
                points += 1
            elif char in "dg":
                points += 2
            elif char in "bcmp":
                points += 3
            elif char in "fhvwy":
                points += 4
            elif char == "k":
                points += 5
            elif char in "jx":
                points += 8
            elif char in "qz":
                points += 10
    return points
```

Notice how all of the letters of the alphabet are accounted for here, and the number of points that each category is worth is on the following line. We'll discuss this overall kind of decision structure in the next chapter, but, at a high level, it is walking through the word character by character and for each character, figuring out which letter it is and adding the appropriate number of points for that letter.

Let's test this to make sure our code is performing the way we want. We know that the word 'zap' should be worth 14 points. And it is:

```
>>> num_points('zap')
14
```

Let's try a couple more. In each case, calculate by hand what you expect the answer to be – that way you'll know if the code is doing the right thing.

```
>>> num_points('pack')
12
```

```
>>> num_points('quack')
20
```

There are many ways to write correct code for a function. If you press <Ctrl>+Enter and look at the Copilot suggestions, you may see different types of code. It doesn't necessarily mean that one of these types is right and the others are wrong.

For example, here's another solution for num_points that we got from the other Copilot suggestions.

```python
points = {'a': 1, 'e': 1, 'i': 1, 'o': 1, 'u': 1, 'l': 1,
          'n': 1, 's': 1, 't': 1, 'r': 1,
          'd': 2, 'g': 2,
          'b': 3, 'c': 3, 'm': 3, 'p': 3,
          'f': 4, 'h': 4, 'v': 4, 'w': 4, 'y': 4,
          'k': 5,
          'j': 8, 'x': 8,
          'q': 10, 'z': 10}
    return sum(points[char] for char in word)
```

Notice here that each letter is listed individually along with its number of points. This structure with the letters and points is called a dictionary and, again, we'll learn about dictionaries properly in a later chapter. The point for now is just that there are often multiple, equivalently good ways for Copilot to write the code for a function. Give this one a try with some test cases before you continue!

3.6.5 The best word

Let's continue with the Scrabble theme. Suppose that Dan has a bunch of words that he can make right now, but he doesn't know which one will give him the most points. Should he make the word 'zap'? 'pack'? 'quack'? It would be nice if we had a function that could take a bunch of words and tell us which is the best one.

How many parameters should we have in such a function? Your first instinct may be 3, one for each of the 3 words in our options. But that wouldn't be very flexible. After all, what if we want to know the best of five words? Ten words? Fifty? We would like our function to work, no matter how many words we are considering.

The trick is to use a function with one (yes, just one!) parameter, which is a *list* of words. Just like numbers and strings and dictionaries, a list is a type supported by Python. It's incredibly useful because it allows us to collect as many values into one place as we like.

We can prompt Copilot like this to get the function that we want:

```
def best_word(word_list):
    """

    word_list is a list of words.

    Return the word worth the most points.
    """
```

How will Copilot know how many points each word is worth? Well, it can call that num_points function that we wrote in the previous section!

Here's the code that Copilot gives us.

```
best_word = ""
    best_points = 0
    for word in word_list:
        points = num_points(word)
        if points > best_points:
            best_word = word
            best_points = points
    return best_word
```

We need to be careful to call this function correctly when testing. This won't work:

```
>>> best_word('zap', 'pack', 'quack')
```

Because that's passing values for three parameters, not one. The right way to call it is to put the words inside square brackets, like this (we'll see more about lists in the next chapter):

```
>>> best_word(['zap', 'pack', 'quack'])
'quack'
```

But, as usual, you shouldn't be content with just one test case. This function should work in other valid but perhaps strange cases, too, such as when we have only one word in the list:

```
>>> best_word(['zap'])
'zap'
```

However, we wouldn't test this function on a list that has no words in it. What would it even make sense to do in that case? Regardless of what the function does, it'd be hard to say one way or the other whether it was doing

the correct thing in a situation where there really is no correct behavior!

Overall, in this chapter we've learned about functions in Python and how we can use Copilot to help us write them. We've also learned about the characteristics of good functions and how important it is to make sure our functions are solving tasks that can be managed well by Copilot. Our next steps in this book all revolve around understanding if the code produced by Copilot is correct and how to fix it when it isn't. In the next chapter, we'll start by learning the basics of being able to read the code produced by Copilot as this gives us the first sanity check for whether Copilot is doing what we think it should be. Then in later chapters we'll dig deeper into how to carefully test the code and what to do when it is wrong.

3.7 Summary

- Problem decomposition involves breaking a large problem into smaller tasks.
- We use functions to perform problem decomposition in our programs.
- Each function must solve one small, well-defined task.
- Functions reduce duplication, make it easier to test our code, and reduce the likelihood of bugs.
- Unit testing involves checking that the function does what we expect on a variety of different inputs.
- A function header or signature is the first line of code of the function.
- Parameters are used to provide information to functions.
- The function header indicates the name of the function and names of its parameters.
- We use the return keyword to output a value from a function.
- A docstring uses the names of each function parameter to describe the purpose of the function.
- To ask Copilot to write a function, we provide it the function header and docstring.
- We get a function to do its work by calling it with values (also called arguments) for its parameters.
- A variable is a name that refers to a value.
- A helper function is a small function written to make it easier to write a bigger function.

- A leaf function doesn't call any other function to do its job.
- To test whether a function is correct, we call it with different types of inputs.
- Every Python value has a type, such as a number, text (string), true/false value (bool) or collection of values (list or dictionary).
- Prompt engineering involves modifying our prompt for Copilot to influence the code that we get back.
- We need to ensure that we import any module (such as string) that our code is using.

3.8 References

[1] J. Sweller. "Cognitive load theory". Psychology of learning and motivation (Vol. 55, pp. 37-76). 2011. Academic Press.

4 Reading Python Code – Part 1

This chapter covers

- Why being able to read code is important
- How to ask Copilot to explain some code
- The top ten programming features that you need to know (part 1)

In Chapter 3, we used Copilot to write several functions for us. What are they good for? Maybe our `money_made` function could be part of a stock trading system. Maybe our `is_strong_password` function could be used as part of a social network website. Maybe our `best_word` function could be used as part of some Scrabble AI. Overall, we can see that we've written some useful functions that could be part of larger programs. And we did this without writing much code ourselves or, indeed, understanding what the code even does.

However, we believe that you need to understand at a high-level what code does and that this will require some time to learn, hence we've split this across two chapters. In this chapter, we'll explain why reading code is important and introduce you to a Copilot labs feature that can help you understand the code. After that, we'll dive into the top ten programming features you'll need to be able to recognize in order to read most basic code produced by Copilot. We'll do the first five in this chapter and the remaining five in the next chapter. Don't worry: you've actually been informally introduced to all ten already—we're just going to deepen your understanding of each one.

4.1 Why we need to read code

When we talk about reading code, what we mean is being able to understand what code does by looking at it. There are two such levels of understanding.

The first is being able to understand, line by line, what a program will do.

This often involves tracing the values of variables as the code runs in order to determine exactly what the code is doing at each step.

The second is being able to determine the overall purpose of a program. As professors, we often test students at this level with questions that ask them to "explain in plain English."

At the end of these two chapters, we'll want you to be able to do both at the level of being able to interpret the code produced by Copilot. We'll start focusing on that line-by-line understanding but toward the end, you'll start being able to look at a small chunk of code and determine its purpose.

Listing 4.1 Best Word function for Scrabble, reprinted from Chapter 3

```
def best_word(word_list):
    """
    word_list is a list of words.

    Return the word worth the most points.
    """
    best_word = ""
    best_points = 0
    for word in word_list:
        points = num_points(word)
        if points > best_points:
            best_word = word
            best_points = points
    return best_word
```

We can illustrate the difference between these two levels of reading code by referring back to our `best_word` function from Chapter 3, reprinted in listing 4.1 above. A **tracing description** of what this program does would be a description of each line. For example, we would say that we're defining a function called `best_word`. We have a variable called `best_word` that we start off as a string with no characters, otherwise known as the empty string. (It's unfortunate that the function and this variable are both called `best_word`, because it makes it trickier to refer to one or the other, but that's what Copilot gave us.) We also have another variable `best_points` that we start at 0. Then we have a `for` loop over each word in the word_list. Inside the `for` loop, we call our `num_points` helper function... and so on. (We'll be explaining how we know what each line of code does over the next two chapters!).

In contrast, a **description of the overall purpose** would be something like our docstring description: "Return the word with the highest Scrabble point value from a list of words." Rather than refer to each line, this description refers to the code's purpose as a whole, explaining what it does at a high level.

You'll come to an overall-purpose level of understanding through a combination of practice with tracing and testing, and we hope you arrive there in full glory by the end of the book. Working at a tracing level generally precedes the ability to work at an overall-purpose level [1], so in this chapter and the next, we're going to focus on the tracing level by understanding what each line of code does.

There are three reasons why we want you to be able to read code.

1. **Helping to determine whether code is correct.** In Chapter 3, we practiced how to test the code that Copilot gives us. Testing is a powerful skill for determining whether code does the right thing, and we will continue to use it throughout the book. But many programmers, the two of us included, will only test something if it seems plausibly correct. If we can determine by inspection that the code is wrong, then we won't bother to test it: we'll try to get the code fixed first. Similarly, we want you to be able to identify when code is simply wrong, without having to spend time testing it. The more code that you can identify as wrong (through quick tracing or honing your overall-purpose skills), the more time you save testing broken code.
2. **Informing testing.** Understanding what the code is doing line by line is useful on its own, but it also helps turbocharge your ability to effectively test. For example, in the next chapter, you'll learn about loops; that they can cause your code to repeat zero times, one time, two times, or as many times as needed. You'll be able to combine that knowledge with what you already know about testing to help you identify important categories of cases to test.
3. **Helping you write code.** We know—you want Copilot to write all of your code! We want that, too. But inevitably, there will be code that Copilot just doesn't get right, no matter how much prompt engineering you do. Or maybe enough prompt engineering could finally cajole

Copilot to write the correct code, but it would be simpler and faster to just do it ourselves. In writing this book, the two of us strive to have Copilot write as much code as possible. But, because of our knowledge of Python programming, we are often able to see a mistake and just fix it without going through any hoops to have Copilot fix it for us. More long-term, we want you to be empowered to learn more programming on your own, and having an understanding of Python is our way to bridge you from this book to other resources later. There is research evidence that being able to trace and explain code is prerequisite to being able to write code [1].

Before we get to it, we need to be clear about the level of depth that we're striving for. We're not going to teach you every nuance of every line of code. Doing so would revert us back to the traditional way that programming was taught prior to tools like Copilot. Rather, through a combination of Copilot tools and our own explanations, we're going to help you understand the gist or overall goal of each line of code. You will need more than this if you endeavor to write large portions of programs in the future. We are trying to target the sweet spot between "this code is magic" and "I know exactly how every line of the code works."

4.2 Asking Copilot to explain code

In Chapter 2, when setting up your computer to use GitHub Copilot, you installed the GitHub Copilot Labs extension to Visual Studio Code. This is an experimental extension that is changing rapidly and is designed to offer new features that are not quite ready for everyday use. We're going to show you one of its best features right now: explaining what Python code does!

We suspect that, soon, the Copilot Labs extension, or parts of it, will be folded into the main Copilot extension. If that happens, the specific steps we give here may vary somewhat, and in that case, we encourage you to consult more general GitHub Copilot documentation.

For now, with the Copilot Labs extension installed, you can highlight some code that you want Copilot to describe to you. Let's try this with our `best_word` function.

Figure 4.1 The Copilot Labs view in VSCode.

∨ EXPLAIN

Explain code   ⌄

▸ Advanced

🐙 Ask Copilot

∨ LANGUAGE TRANSLATION

Translate code into

⌄

∨ BRUSHES

☺	‹T›	🐞	▷	🗑	☰	⛨
READABLE	ADD TYPES	FIX BUG	DEBUG	CLEAN	LIST STEPS	MAKE ROBUST

◔   📖   🎨

∨ TEST GENERATION

Selected Text

⚠ Test generation is currently only supported for JavaScript and TypeScript.

First, click on the Copilot Labs tab in your Activity Bar (on the left-hand side of VSCode) and you should see a window similar to figure 4.1.

Figure 4.2 The code from the best_word function highlighted in the editor.

```python
def best_word(word_list):
    """

    word_list is a list of words.


    Return the word worth the most points.
    """
    best_word = ""
    best_points = 0
    for word in word_list:
        points = num_points(word)
        if points > best_points:
            best_word = word
            best_points = points
    return best_word
```

Next, highlight all of the code for our best_word function as is highlighted in figure 4.2. (You may need to have Copilot generate the code for you again if you didn't save it from Chapter 3.)

Figure 4.3 The code from the best_word function appearing in Copilot Labs.

∨ **EXPLAIN**

```python
def best_word(word_list):
    """

    word_list is a list of words.

    Return the word worth the most points.
    """
    best_word = ""
    best_points = 0
    for word in word_list:
        points = num_points(word)
        if points > best_points:
            best_word = word
            best_points = points
    return best_word
```

Explain code   ∨

▸ Advanced

🐢 Ask Copilot

After highlighting the code, you should now see the code appear in the left in the EXPLAIN feature as shown in figure 4.3

Figure 4.4 The different options for explaining your code in Copilot Labs.

∨ EXPLAIN

```python
def best_word(word_list):
    """

    word_list is a list of words.

    Return the word worth the most points.
    """
    best_word = ""
    best_points = 0
    for word in word_list:
        points = num_points(word)
        if points > best_points:
            best_word = word
            best_points = points
    return best_word
```

Explain code   ∨

Custom
Explain code
Code does following
Code does following (English)
Show example code

Figure 4.4 shows the different prompts provided by Copilot that you can use to ask for code explanations. Each will generally yield different responses that vary in how specific they are and whether they produce fewer or more examples. We'll leave it at the default prompt of "Explain Code," but if you like, you can try other prompts from the drop-down box (shown in figure 4.4). The current options are "Explain Code," "Code Does Following," "Code Does Following (English)," and "Show Example Code." There's also a "Custom" option that allows you to use whatever prompt you like.

The last step is to click Ask Copilot (as shown as the bottom button in figure 4.3). You'll get an explanation of the code. As usual, Copilot is nondeterministic, so your results will likely differ from ours. In fact, if an explanation is confusing you, you might try clicking Ask Copilot again or changing the prompt to get a different explanation. As educators, this is the start of a dream come true, where students will be able to ask for as many explanations as they need to fully understand how code works.

COPILOT lab explanations can be wrong

As with the code produced by Copilot, the explanations from Copilot Labs can be wrong too. You might be thinking, "Wait, what's the value of a tool to understand code if it can be wrong?!" First, let us say that we agree, this can be really frustrating. However, it is right often enough in our experience that it can be really helpful, particularly for understanding code basics like the things we're teaching in these two chapters. Moreover, what students do now when they are confused about their code is to ask friends or websites about the code and, candidly, those make mistakes too. A way to help mitigate the potential for error is to ask Copilot for its explanation a couple times as you are less likely to get grossly incorrect answers across multiple tries.

Here's what Copilot gave us the first time we asked:

```
""" Here is the explanation for the code above:
1. We define a function called best_word that takes one argument,
2. We define the variables best_word and best_points and set them
3. We use a for loop to iterate over the elements in word_list.
4. We define a variable called points that is equal to the result
5. We use an if statement to check if points is greater than best
6. If it is, we set best_word equal to word and best_points equal
```

```
7. Finally, we return best_word. """
```

To us, this is an excellent description of what the code does line by line. Notice, however, in order to make full sense of it, you need to know important Python terminology and features. The explanation from Copilot above refers to functions, arguments, variables, strings, for loops, if statements, and returning. You know much of this terminology already, but if you did not, then this explanation would be much less useful. By the end of this and the next chapter, you will know the main terminology and features of Python that you need to interpret a typical explanation from Copilot.

4.3 The top ten programming features you need to know, pt. 1

We're going to give you a whirlwind tour of the ten Python features that you'll need for the rest of your programming journey, starting with the first five of those in this chapter.

Python is an interactive language, which makes it easier than other languages for us to play around and see what stuff does. We'll take advantage of that here as we explore programming features. This is how the two of us learned Python, and indeed how many thousands of programmers have done so. Don't hesitate to experiment! To get started, press Ctrl+Shift+P and type REPL, then select Python: Start REPL. This should result in the situation shown in figure 4.5.
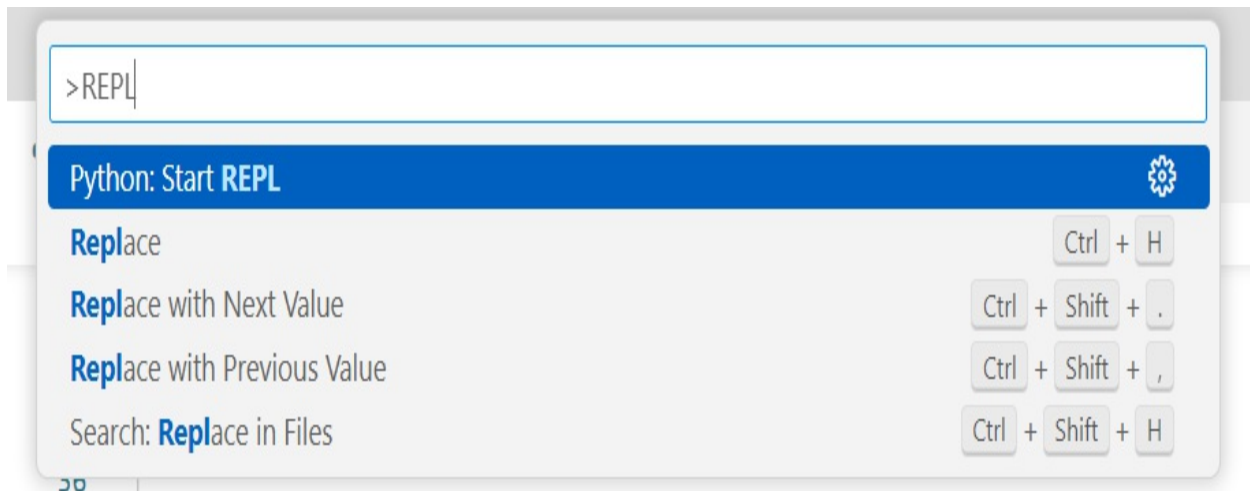
Figure 4.5 Starting REPL from VSCode.

Figure 4.6 REPL running in VSCode.

This will put you back at the same Python prompt as in Chapter 3 (as shown in figure 4.6), except with none of your functions loaded. Then we can start typing Python code. For example, type

```
>>> 5 * 4
```

And press ENTER. You'll see the response of 20. We won't spend time on simple math here, but the way you interact with Python to learn how it works is exactly the same: you type some code and Python responds.

#1. Functions

You learned all about functions in Chapter 3, so let's just summarize what we

learned.

You use functions to break a large problem into smaller pieces. In retrospect, that `best_word` function that we wrote in Chapter 3 is a pretty big task: it has to figure out which word in a list of words is worth the most points. How many points is a word worth? Aha—that's a subtask that we can carve out from this function. And, indeed, we had done that in our earlier `num_points` function.

We design a function to take parameters, one parameter for each piece or collection of data that the function needs to do its job. After doing their work, most functions use return to send the answer back to the line of code that called them. When we call a function, we pass values, known as arguments, with one value for each parameter; and we often store that return value using a variable.

For each program we write, we'll likely need to design a few functions—but there are also some functions that are built-in to Python that we get for free. We can call those like we call our own functions. For example, there's a built-in max function that takes one or more arguments and tells us the biggest:

```
>>> max(5, 2, 8, 1)
8
```

There's also the input function that was used in our `get_strong_password` function from chapter 3. It takes an argument that becomes the prompt, and it returns whatever the user types at the keyboard:

```
>>> name = input("What is your name? ")
What is your name? Dan
>>> name
'Dan'
```

If input is the function to receive input from the keyboard, is there an output function to output a message to the screen? Well, yes, but it's called print, not output:

```
>>> print('Hello', name)
Hello Dan
```

#2. Variables

A variable is a name that refers to a value. We used variables in Chapter 3 to keep track of return values from functions. We also just used a variable here to hold the user's name. Whenever we need to remember a value for later, we use a variable.

To assign a value to a variable, we use the = (equal sign) symbol, which is called the **assignment** symbol. It figures out the value of whatever is on the right, then assigns that to the variable:

```
>>> age = 20 + 4 #A
>>> age
24
```

THE = SYMBOL IS DIFFERENT IN PYTHON than in math

The = sign is used in Python, and other programming languages, to denote **assignment**. The variable on the left side of the equal symbol is given the value of the calculation performed on the right-hand side of the equal symbol. This is *not* a permanent relationship as the variable can have its value changed. People new to programming who are strong in math can find this confusing, but just remember that the = sign in Python means assignment, not equality.

We can use the variable in a larger context, called an **expression**. The value that the variable refers to gets substituted for its name:

```
>>> age + 3 #A
27
>>> age #B
24
```

VARIABLES PERSIST IN THE python prompt

We assigned age in the earlier batch of code. Why can we keep referring to it? Any variable declared during a session of programming with your Python prompt will stick around until you quit. That's just how variables work in programs, too: they're available as soon as you assign a value to them.

But notice that the variable `age` didn't change when we said `age+3`! To change it, we need another = assignment statement:

```
>>> age = age + 5 #A
>>> age
29
```

Let's see a few more ways to change what a variable refers to. We'll include some explanations as annotations with the code.

```
>>> age += 5  #A
>>> age
34
>>> age *= 2  #B
>>> age
68
```

#3. Conditionals

Whenever our program has to make a decision, we need a conditional statement. For example, back in Chapter 2, we needed to make a decision about which quarterbacks to include in our data. To do so, we used `if` statements.

Remember our larger function from Chapter 3? We've reproduced it here as listing 4-2.

Listing 4.2 Function to determine the larger of two values, reprinted from Chapter 3

```
def larger(num1, num2):
    if num1 > num2:      #A
        return num1      #B
    else:                #C
        return num2      #D
```

The `if-else` structure in listing 4-2 is known as a **conditional** statement, and it allows our program to make decisions. Here, if `num1` is greater than `num2`, then `num1` is returned; else, `num2` is returned. That's how it returns the larger one!

After the word `if`, we put a Boolean condition (`num1 > num2`). A Boolean

condition is an expression that tests a condition where the result would either be `True` or `False`. If it's `True`, then the code under the `if` runs; if it's `False`, then the code under the `else` runs. We create Boolean expressions using comparison symbols such as >= for greater than or equal to, < for less than, and != for not equal to. Notice that we're using indentation not only for the code of the function but also for the code of the `if` and `else` parts of the `if-else` statement. Indentation is necessary for the code to function properly, so this is worth paying attention to (in the next chapter, we talk more about indentation). This is how Python knows which lines of code belong to the function and which additionally belong to the `if` or `else`.

We can play around with conditional statements at the Python prompt, too—we don't need to be writing code inside of a function. Here's an example:

```
>>> age = 40 #A
>>> if age < 40: #B
...     print("Binging Friends")
... else: #C
...     print("What's binging?")
...
What's binging?
```

You'll notice that the prompt changes from >>> to ... when you're typing inside the if-statement. The change of prompt lets you know that you're in the middle of typing some code that you need to complete. You need an extra press of enter when you're done with the else code to get out of the ... prompt and back to the >>> prompt.

We set the age variable to 40. As `40 < 40` is `False`, the `else` runs.

Let's try again, this time making the `if` run:

```
>>> age = 25      #A
>>> if age < 40: #B
...     print("Binging Friends")
... else:         #C
...     print("What's binging?")
...
Binging Friends
```

You might see some if-statements with no `else` part, and that's okay: the

`else` part is optional. In that case, if the condition is `False` then the if-statement won't do anything:

```
>>> age = 25 #A
>>> if age == 30: #B
...     print("You are exactly 30!")
...
```

Notice that the way to test whether two values are equal is to use two equal signs, ==, not one equal sign. (We already know that one equal sign is for the assignment statement to assign a value to a variable)!

What do you do if you have more than two possible outcomes? For example, let's say that people's age determines the show that they'll likely binge, like this:

Table 4.1 Possible favorite TV shows by age

Age	Show
30-39	Friends
20-29	The Office
10-19	Pretty Little Liars
0-9	Chi's Sweet Home

We can't capture all these outcomes with just an if/else, so the `elif` (short for else if) allows us to capture the logic for more than 2 possible outcomes as shown in the code below. We're presenting this without the python prompt (`>>>`) and `...` since this would be a lot to type in!

```
if age >= 30 and age <= 39: #A
    print("Binging Friends")
elif age >= 20 and age <= 29: #B
    print("Binging The Office")
elif age >= 10 and age <= 19:
    print("Binging Pretty Little Liars")
elif age >= 0 and age <= 9:
    print("Binging Chi's Sweet Home")
else: #C
    print("What's binging?") #C
```

We're using and to capture a complex condition. For example, in the first line, we want age to be greater than or equal to 30 and less than or equal to 39. Python works from top to bottom, and when it finds a condition that's true, it runs the corresponding indented code. Then it stops checking any remaining elifs or else—so if two conditions happened to be true, only the code for the first one would run.

Try experimenting with various values for the age variable to observe that the correct code runs in each case. In fact, if we were serious about testing this code, we could use the if-statement structure for a good sense of the values we'd want to test. It's all about testing on the boundaries of values. For example, we definitely want to test the ages 30 and 39 to make sure, for example, that we're correctly capturing the full 30-39 range with the first condition. Similarly, we'd want to test 20, 29, 10, 19, 0, 9, and then something larger than 39 to test the else way at the bottom.

If you use additional ifs rather than elifs, then they become separate if statements, rather than a single if statement. This matters, because Python always checks each independent if statement on its own, regardless of what may have happened in above if statements.

For example, let's change the elifs to ifs in our age code. That gives us the following:

```
if age >= 30 and age <= 39:
    print("Binging Friends")
if age >= 20 and age <= 29: #A
    print("Binging The Office")
if age >= 10 and age <= 19: #B
    print("Binging Pretty Little Liars")
```

```
if age >= 0 and age <= 9: #C
    print("Binging Chi's Sweet Home")
else: #D
    print("What's binging?")
```

Suppose that you put `age = 25` above this code and run it. What do you think will happen?

Well, the second `if` condition, `age >= 20 and age <= 29`, is True, so we'll certainly output `Binging The Office`. But that's not all that happens! Remember: because we're using `if`s here, each of the remaining ones is going to be checked. (If they were `elif`s, we'd be done.) `age >= 10 and age <= 19` is False, so we're not going to output `Binging Pretty Little Liars`.

The final `if` condition, `age >= 0 and age <= 9`, is also False, so we're not going to output `Binging Chi's Sweet Home`. But this `if` has an `else`! So we **are** going to output `What's binging?`. We didn't intend this! We only wanted `What's binging?` for people over 40. This is all to say that `if` and `elif` behave differently, and that we need to be using the one that matches the behavior that we want (`if` if we want multiple chunks of code to potentially run, `elif` if we want only one).

#4. Strings

As we learned in Chapter 3, a string is the type that we use whenever we want to store text. Text is everywhere – stats like in Chapter 2, passwords, books – so strings show up in almost every Python program.

We use quotation marks to indicate the beginning and end of the string. You'll see Copilot use double quotes or single quotes. It doesn't matter which you use, just be sure to start and end the string with the same type of quote.

Strings come with a powerful set of methods. A **method** is a function that's associated with a particular type—in this case, strings. The way you call a method is a little different than how you call a function: we need to put the value on which we're calling the method first, outside of the parentheses, followed by a dot.

In Chapter 3, Copilot used some string methods to implement `is_strong_password`. Let's try using those methods here to gain a better understanding of how they work:

```
>>> 'abc'.isupper()   #A
False                 #A
>>> 'Abc'.isupper()   #A
False                 #A
>>> 'ABC'.isupper()   #A
True                  #A

>>> 'abc'.isdigit()    #B
False                  #B
>>> '345bc'.isdigit()  #B
False                  #B
>>> '345'.isdigit()    #B
True                   #B

>>> 'abc6'.isalnum()       #C
True                       #C
>>> 'abc def'.isalnum()  #C
False                    #C
>>> 'abcdef#'.isalnum()  #C
False                    #C
```

There are many other string methods as well. Some help you convert strings between lowercase and uppercase, others help you search a string, and still others help you format a string (for example, centering or right-justifying it).

Copilot's Explain feature can often be helpful if you're not sure what a given string method does. For example, what does this function do?

```
def clean_number(phone_number):
    phone_number = phone_number.replace('(', '') #A
    phone_number = phone_number.replace(')', '') #B
    phone_number = phone_number.replace('-', '') #C
    return phone_number
```

The first time we asked Copilot to explain, we got this:

```
""" Here is the explanation for the code above:
1. The replace() method returns a copy of the string in which the
2. When we call the replace() method and pass in the value of the
3. In the above code, we are calling the replace() method three t
```

This is all accurate information, but one thing it fails to answer is: are we getting rid of every (,), and - in the phone number, or only the first of each? It turns out that the replace method replaces them all! So, if a phone number had multiple parentheses or dashes, they would all be gone in the returned phone number. The reason that they're gone is because we're replacing each with the empty string, and the empty string has no characters. To learn about this nuance of the replace method, you could conduct your own testing using replace at the Python prompt.

Other times, we received very general information about what a function does and what returning a value does—not very helpful in this specific case.

Here's another explanation that we got:

```
""" Here is the explanation for the code above:
1. First we convert the phone_number argument into a string
2. Then we replace the parentheses and dashes from the string wit
3. Then we return the string with the parentheses and dashes remo
```

Points 2 and 3 are accurate but point 1 is wrong. We aren't converting anything to a string! This is an example of why we need a baseline of Python knowledge: to help us separate truth from bits of incorrect information.

You'll also often see what look like mathematical operators being used on strings. They are the same as math symbols, but they do different things on strings. The + operator is used to put two strings together, and the * operator is used to repeat a string a specific number of times. For example:

```
>>> first = 'This is a '
>>> second = 'sentence.'
>>> sentence = first + second  #A
>>> print(sentence)
This is a sentence.
>>> print('-=' * 5)  #B
-=-=-=-=-=
```

#5. Lists

A string is great when we have a sequence of characters, like a password or a single Scrabble word. But sometimes we need to store many words or many

numbers. For that, we need a list.

We used a list in Chapter 3 for the `best_word` function, because that function needed to work with a list of individual words.

Whereas we use quotation marks to start and end a string, we use opening and closing square brackets to start and end a list. And, as for strings, there are many methods available on lists. To give you an idea of the kinds of list methods available and what they do, let's explore some of these below:

```
>>> books = ['The Invasion', 'The Encounter', 'The Message'] #A
>>> books
['The Invasion', 'The Encounter', 'The Message']
>>> books.append('The Predator')  #B
>>> books
['The Invasion', 'The Encounter', 'The Message', 'The Predator']
>>> books.reverse() #C
>>> books
['The Predator', 'The Message', 'The Encounter', 'The Invasion']
```

Figure 4.7 List elements can be accessed through either positive or negative indexes.

Positive Index	Negative Index	books
0	-4	"The Predator"
1	-3	"The Message"
2	-2	"The Encounter"
3	-1	"The Invasion"

Many Python types, including strings and lists, allow you to work with particular values using an **index**. The index starts at 0 and goes up to, but not including, the number of values. That is, the first value has index 0 (not index 1!), the second has index 1, and so on. The last value in the list is at the index that's the length of the list minus 1. The length of the list can be determined by using the `len` function. For example, if we do `len(books)`, we'll get a value of 4 (so the valid indices are from 0 up to and including 3). People also often use negative indices, which gives another way to index each value: the rightmost value has index -1, the value to its left has index -2, and so on. Figure 4.7 depicts this example with both positive and negative indexing.

Let's practice indexing on the current books list:

```
>>> books
['The Predator', 'The Message', 'The Encounter', 'The Invasion']
>>> books[0]   #A
'The Predator'
>>> books[1]
'The Message'
>>> books[2]
'The Encounter'
>>> books[3]
'The Invasion'
>>> books[4]   #B
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> books[-1]   #C
'The Invasion'
>>> books[-2]
'The Encounter'
```

There's also a way to pull multiple values out of a string or list, rather than just one. It's called slicing. We specify the index of the first value, a colon, and the index to the right of the value, like this:

```
>>> books[1:3] #A
['The Message', 'The Encounter']
```

We specified 1:3, so you might expect to get the values including index 3. But the value at the second index (the one after the colon) is not included. It's counterintuitive, but true!

If we leave out the starting or ending index, Python uses the start or end as appropriate:

```
>>> books[:3] #A
['The Predator', 'The Message', 'The Encounter']
>>> books[1:] #B
['The Message', 'The Encounter', 'The Invasion']
```

We can also use indexing to change a specific value in a list. For example:

```
>>> books
['The Predator', 'The Message', 'The Encounter', 'The Invasion']
>>> books[0] = 'The Android' #A
>>> books[0]
'The Android'
>>> books[1] = books[1].upper() #B
>>> books[1]
'THE MESSAGE'
>>> books
['The Android', 'THE MESSAGE', 'The Encounter', 'The Invasion']
```

If we try that on a string, though, we get an error:

```
>>> title = 'The Invasion'
>>> title[0]  #A
'T'
>>> title[1]
'h'
>>> title[-1]
'n'
>>> title[0] = 't'  #B
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

A string is known as an **immutable** value, which means that you cannot change its characters. You can only create an entirely new string. By contrast, a list is known as a **mutable** value, which means that you can change it.

In this chapter, we introduced you to five of the most common code features in Python. We'll continue with five more in the next chapter. We also showed you how you can use the Copilot explanation tool to help you understand what the code is doing and offered guidance for verifying the veracity of

these explanations.

Table 4.2 Summary of Python Code Features from this Chapter

Code Element	Example	Brief Description
Functions	`def larger(num1, num2)`	Code feature that allows us to manage code complexity. Functions take in inputs, process those inputs, and possibly return an output.
Variables	`age = 25`	A human readable name that refers to a stored value. It can be assigned using the = assignment statement

Conditionals	`if age < 18:`
	`print("Can't vote")`
	`else:`
	`print("Can vote")`
		Conditionals allow the code to make decisions. In Python, we have three key words associated with conditionals: if, elif, and else.

Strings	`name = 'Dan'`	Strings store a sequence of characters (text). There are many powerful methods available for modifying strings.
	`list = ['Leo', 'Dan']`	A sequence of values of any type. There are many powerful

Lists		methods available for modifying lists.

4.4 Summary

- We need to be able to read code in order to determine whether it is correct, test effectively, and write our own code when needed.
- The Copilot Labs Extension can provide line by line explanations of code to explain to you what the code is doing.
- Python has built-in functions such as max, input, and print that we call just like we call our own functions.
- A variable is a name that refers to a value.
- An assignment statement makes a variable refer to a specific value.
- An if-statement is used to have our programs make decisions and proceed down one of multiple paths.
- A string is used to store and manipulate text.
- A method is a function associated with a particular type.
- A list is used to store and manipulate a general sequence of values (like a sequence of numbers or a sequence of strings).
- Each value in a string or list has an index; indexing starts at 0, not 1.
- Strings are immutable (not changeable); lists are mutable (changeable).

4.5 References

[1] R. Lister, C. Fidge, and D. Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In Proc. of 2020 ACM Conference on Innovation and Technology in Computer Science Education, July 2009, pp. 161–165.