4 Reading Python code: Part 1

This chapter covers

- Why knowing how to read code is important
- How to ask Copilot to explain code
- Using functions to break down large problems
- Using variables to hang on to values
- Using if-statements to make decisions
- Using strings to store and manipulate text
- Using lists to collect and manipulate many values

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, our `is_strong_password` function could be used as part of a social networking website, or our `best_word` function could be used as part of some Scrabble AI. Overall, 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. Because this will require some time to learn, we've split this discussion over two chapters. In this chapter, we'll explain why reading code is important and introduce you to a Copilot feature that can help you understand the code. After that, we'll dive into the top 10 programming features you'll need to recognize 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 10 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 understanding what code does by looking at it. There are two such levels of understanding, and they're both important.

The first level 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 to determine exactly what the code is doing at each step. This is useful, but isn't the whole story. It's like following a cooking recipe step-by-step without knowing what the recipe is making or how its ingredients fit together. At the end, you might have a cake, but you might not be able to grasp why the cake turned out so well or how you could come up with such a recipe yourself. You need to understand why the recipe, as a whole, is effective.

So, that's the second level: determining the overall purpose of a program. That is, you need to be able to zoom out, away from each line, and understand what the program is doing at a high level.

At the end of these two chapters, we want you to be able to begin to understand code from Copilot at both of these levels. We'll start focusing on that line-by-line understanding, but toward the end of this chapter and through the rest of the book, you'll start being able to look at a small chunk of code and determine its overall purpose.

We can illustrate the difference between the two levels of reading code by referring back to our `best_word` function from chapter 3, reprinted in the following listing.

Listing 4.1 `best_word` function for Scrabble

```
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
```

A *tracing description* of what this program does is a description of each line. For example, we would say that we're defining a function called `best_word` that takes one parameter called `word_list`. 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 store its return value in the points variable, and so on. (We'll explain how we know what each line of code does over this chapter and the next.)

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. *To help 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'll 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 fix the code first. Similarly, we want you 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. *To inform testing* —Understanding what the code is doing line by line is useful on its own, but it also helps turbocharge your ability to test effectively. 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. *To help 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're often able to see a mistake and just fix it without going through any hoops to have Copilot fix it for us. Longer term, we want you to be empowered to learn more programming on your own, and having an understanding of Python is our way to provide a bridge for 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 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'll need more than this if you endeavor to write large portions of programs in the future. We're 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

To this point, we’ve used Copilot to generate code for us. That’s its most well-known feature, but it’s not the only one. We’re going to show you another of its best features right now: explaining what Python code does!

The Copilot extension is always changing. The specific steps we give here may vary somewhat, and, in that case, we encourage you to consult more general GitHub Copilot documentation.

To try the Explain feature, you need to highlight some code that you want Copilot to describe to you. Let’s try this with our `best_word` function (listing 4.1). If you don’t have this code typed in from chapter 3, please enter it now.

figure

Figure 4.1 The code from the `best_word` function highlighted in the editor

Highlight the code as in figure 4.1. After this, we have a few options. The first is to click the Chat icon in the Activity Bar (on the left-hand side of VS Code), or press Ctrl-Alt-I. This will open the Copilot Chat interface, as in figure 4.2. (There are other interfaces for Chat, including a Quick Chat that you can open with Ctrl-Shift-I—it doesn’t matter which chat interface you use.) Copilot Chat is able to respond to your questions not only with natural language but also with blocks of code, buttons/links that you can click to invoke commands, and more. It also suggests questions that you may wish to ask and, of course, allows you to type whatever code-related questions you like.

figure

Figure 4.2 Copilot Chat interface in VS Code

For now, we'll use Copilot Chat to explain the code. In the text box, type the following command:

```
/explain
```

If Copilot asks, you want the @workspace option. If this doesn't work for you, you can also try this command:

```
/explain how the selected code works
```

Typing a forward slash (/) is how you tell Copilot Chat that you're specifying a command (Copilot documentation refers to these as slash commands), rather than natural language. In this case, we're asking for the `/explain` command to be run by Copilot for the selected text.

Alternatively, you could ask Copilot to explain the code by highlighting the code in the function just like in figure 4.1. Then, right-click the code in the function, and you'll see the menu that appears in figure 4.3.

figure

Figure 4.3 The menu that appears when you right-click the selected `best_word` function

In this menu, highlight Copilot, and you'll see the list of options shown in figure 4.4. These options are the more commonly used features with Copilot.

figure

Figure 4.4 A list of options that you can use with Copilot

The last step is to select the Explain This option. The Copilot Chat window will open (as shown earlier in figure 4.2), and 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 the `/explain` command again or writing your own prompt to get a different explanation. As educators, this is the start of a dream come true, where learners will be able to ask for as many explanations as they need to fully understand how code works.

Copilot explanations can be wrong

As with the code produced by Copilot, the explanations from Copilot 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, we agree that this can be really frustrating. However, Copilot’s explanations are right often enough in our experience that they can be really helpful, particularly for understanding code basics such as the things we’re teaching in these two chapters. Moreover, what learners historically did when they were confused about their code was to ask friends or websites about the code, which can lead to mistakes too. A way to help mitigate the potential for error is to ask Copilot for its explanation a couple of times because you’re less likely to get grossly incorrect answers across multiple tries.

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

The provided Python code defines a function named `best_word` that takes a list of words as an argument (`word_list`). The purpose of this function is to determine and return the word from the list that is worth the most points.

The function starts by initializing two variables: `best_word` and `best_points`. `best_word` is an empty string that will eventually hold the word with the highest point value, while `best_points` is set to 0 and will hold the highest point value found.

The function then enters a loop where it iterates over each word in the `word_list`. For each word, it calculates the point value of the word by calling the `num_points` function (not shown in the provided code). The `num_points` function presumably takes a word as an argument and returns its point value.

If the point value of the current word is greater than the current `best_points`, the function updates `best_word` to be the current word and `best_points` to be the points of the current word. This way, the function always keeps track of the word with the highest point value.

After the loop has iterated over all the words in the list, the function returns `best_word`, which at this point should be the word with the highest point value from the list. If the list is empty or all words have a point value of 0, the function will return an empty string.

To us, this is an excellent description of what the code does. It starts with an overall explanation of the code, and then proceeds to give more of a line-by-line explanation. Notice, however, to make full sense of it, you need to know important Python terminology and features. The explanation from Copilot refers to functions, arguments, variables, strings, loops, making decisions ("If the point value"), and returning. This explanation is quite useful if you understand the terminology, but much less useful if you don't. By the end of this and the next chapter, you'll know the main terminology and features of Python that you need to interpret a typical explanation from Copilot.

Asking for explanations

The type of explanation that you get from the `/explain` command may not meet your requirements. For example, suppose that you want Copilot to explicitly give you each line of code and explain what it does, so that you can more easily connect the code to its specific explanation. You can make that happen! Try this prompt: "Explain that code again, but do it line by line. Give each line of code and the explanation of what it does." Doing so gave us a very detailed breakdown of each line of code, followed by its explanation. We encourage you to craft your own prompts if built-in commands aren't giving you the level of detail that you want.

4.3 Top 10 programming features you need to know: Part 1

We’re going to give you a whirlwind tour of the 10 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 with 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 how many thousands of programmers have done so. Don’t hesitate to experiment! To get started, press Ctrl–Shift–P and type `REPL`, and then select Python: Start REPL. This should result in the situation shown in figure 4.5. (REPL stands for read-execute-print-loop. It’s called that because Python reads what you type, executes/runs it, prints the results back to you, and does all of this over and over in a loop.)

figure

Figure 4.5 Starting REPL from VS Code

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.

figure

Figure 4.6 REPL running in VS Code

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. Now, let’s jump into the first five of those 10 Python features we mentioned.

4.3.1 #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 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, which we did 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 largest:

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

There's also the `input` function, which we used in our `get_strong_password` function from chapter 3. It takes an argument that becomes the prompt to the user, 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
```

4.3.2 #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 = (equals sign) symbol, which is called the *assignment* symbol. It figures out the value of whatever is on the right and then assigns that to the variable:

```
>>> age = 20 + 4        #1
>>> age
24
```

#1 The right-hand side of the = symbol is evaluated, which means 20 + 4 is evaluated to be 24. Then, the variable age is assigned the value of 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 = symbol is given the value of the calculation performed on the right side of the = 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         #1
27
>>> age       #2
24
```

#1 Age is still available in the Python prompt and has the value 24. 24 + 3 is evaluated to be 27.
#2 The expression of age + 3 doesn't change age because we didn't reassign age.

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      #1
>>> age
29
```

#1 We've changed age by doing an assignment (the = symbol).

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       #1
>>> age
34
>>> age *= 2       #2
>>> age
68
```

#1 A shortcut way to add. age += 5 is equivalent to age = age + 5.
#2 A shortcut way to multiply by 2. age *= 2 is equivalent to age = age * 2.

4.3.3 #3. Conditionals

Whenever our program has to make a decision, we need a conditional statement. For example, in chapter 2, we needed to make a decision about which players to include in our data (we wanted only quarterbacks). To do so, we used `if` statements.

Remember our larger function from chapter 3? We've reproduced it here in the following listing.

Listing 4.2 Function to determine the larger of two values

```
def larger(num1, num2):
    if num1 > num2:     #1
        return num1           #2
    else:                    #3
        return num2     #4
```

#1 The num1 > num2 expression will be True if num1 is greater than num2 and will be False if num1 is less than or equal to num2.
#2 This line is executed when num1 is greater than num2.
#3 The else keyword must be paired with an if keyword. When the if doesn't execute (num1 > num2 is False), the else executes instead.
#4 This line is executed when num1 isn't greater than num2.

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; otherwise, `num2` is returned. That's how it returns the larger one!

After `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, > for greater than, == for equal to, 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 it's worth paying attention to (we talk more about indentation in the next chapter). 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      #1
>>> if age < 40:                          #2
...     print("Binging Friends")
... else:                              #3
...     print("What's binging?")
...
What's binging?
```

#1 We assign 40 to age.
#2 Because age is 40, this code is asking whether 40 < 40. It's not, so the if part of the code is skipped.
#3 The else portion runs because the if condition is False.

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 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             #1
>>> if age < 40:                         #2
...     print("Binging Friends")
... else:                              #3
...     print("What's binging?")
...
Binging Friends
```

#1 We assign 25 to age.
#2 Because age is 25, this is asking whether 25 < 40. It is, so the if part of the code runs.
#3 The else portion doesn't run (we already ran the if part of the code).

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            #1
>>> if age == 30:              #2
...     print("You are exactly 30!")
...
```

#1 We assign 25 to age.
#2 == tests to see if the two values are equal.

Notice that the way to test whether two values are equal is to use two equals signs, (`==`), not one equals sign. (We already know that one equals 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 a person's age determines the show they'll likely binge, as shown in table 4.1.

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 of these outcomes with just an `if-else`, so the `elif` (short for `else-if`) allows us to capture the logic for more than two possible outcomes, as shown in the following code. We're presenting this code without the Python prompts (>>> and ...) because this would be a lot to type:

```
if age >= 30 and age <= 39:       #1
    print("Binging Friends")
elif age >= 20 and age <= 29:       #2
    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:
    print("What's binging?")      #3
```

#1 This is True if both age >= 30 and age <= 39 are true; for example, if age were 35.
#2 This condition is checked if the above condition is False.
#3 This code runs if all conditions above are False.

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 the remaining `elif`s 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 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 `if`s rather than `elif`s, 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 previous `if` statements.

For example, let's change the `elif`s to `if`s in our age code. That gives us the following:

```
if age >= 30 and age <= 39:
    print("Binging Friends")
if age >= 20 and age <= 29:                    #1
    print("Binging The Office")
if age >= 10 and age <= 19:
    print("Binging Pretty Little Liars")
if age >= 0 and age <= 9:
    print("Binging Chi's Sweet Home")
else:                              #2
    print("What's binging?")
```

#1 This condition is always checked.
#2 This else goes with the most recent if statement.

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 whose age is at least 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, and `elif` if we want only one).

4.3.4 #4. Strings

As we learned in chapter 3, a string is the type 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 from how you call a function. When you call a function, you put the argument values in parentheses. With methods, we still put values in parentheses, but we need to put the value on which we're calling the method first, outside of the parentheses, followed by a dot. We'll see an example of that next!

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()      #1
False
>>> 'Abc'.isupper()
False
>>> 'ABC'.isupper()
True
>>> 'abc'.isdigit()        #2
False                      #2
>>> '345bc'.isdigit()      #2
False                      #2
>>> '345'.isdigit()        #2
True                       #2

>>> 'abc6'.isalnum()         #3
True                         #3
>>> 'abc def'.isalnum()      #3
```

```
False                        #3
>>> 'abcdef#'.isalnum()      #3
False                        #3
```

#1 isupper needs all characters to be uppercase for it to return True.
#2 isdigit needs all characters to be numbers for it to return True.
#3 isalnum needs all characters to be alphanumeric to return True (a space isn't considered alphanumeric).

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 (e.g., 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('(', '')     #1
    phone_number = phone_number.replace(')', '')      #2
    phone_number = phone_number.replace('-', '')     #3
    return phone_number
```

#1 Removes all (characters from phone number
#2 Removes all) characters from phone number
#3 Removes all – characters from phone number

Remember that we can ask Copilot to explain code, as we demonstrated in section 4.2. Here's what we got when we asked Copilot to explain this code:

The provided Python code defines a function named `clean_number`. This function is designed to take a single argument, `phone_number`, which is expected to be a string representing a phone number. The purpose of this function is to remove certain characters from the phone number string, specifically parentheses "(" and ")" and hyphens "-".

The function uses the `replace()` method, which is a built-in method in Python for strings. The `replace()` method replaces a specified phrase with another specified phrase. It takes two arguments: the first is the substring that needs to be replaced, and the second is the substring that will replace the first substring.

In this function, the `replace()` method is called three times on `phone_number`. Each call to `replace()` is used to remove one type of character: first the open parenthesis "(", then the close parenthesis ")", and finally the hyphen "-". It does this by replacing these characters with an empty string "", effectively removing them.

After all replacements are done, the cleaned `phone_number` is returned by the function. This will be a string containing the digits of the phone number without any parentheses or hyphens.

This function could be useful in situations where you need to process or store phone numbers in a consistent, character-free format. For example, you might want to remove these characters when storing phone numbers in a database, or when comparing phone numbers for equality.

That confirms what this function does: removes parentheses and hyphens from a phone number string. And it uses the `replace` method to do so.

Wait—"replace" sounds like we're changing some characters to some other characters (replacing them), not removing them. How is `replace` able to remove characters for us, then? The explanation said, "It does this by replacing these characters with an empty string "", effectively removing them." That's a clue to how `replace` is doing this, but we might want to know more. We therefore next used the prompt "How does the replace method use the empty string to remove characters?" and received the following explanation from Copilot:

The `replace()` method in Python replaces all occurrences of a specified substring in the string it's called on with another specified substring.

In this case, the specified substrings are the characters "(", ")", and "-". The method is called on the `phone_number` string, and the second argument to `replace()` is an empty string "".

So, what happens is that every occurrence of the specified character in `phone_number` is replaced with an empty string. This effectively removes that character from the string, because replacing something with nothing is equivalent to removing it.

This is done three times in the `clean_number` function, once for each character that needs to be removed. After all three characters have been removed, the cleaned `phone_number` is returned.

And if you wanted to learn even more, you could continue asking Copilot questions or do your own testing at the Python prompt. For example, you could verify that `replace` removes all (,), and - characters, even if there are multiples of each. And you could explore using `replace` to actually replace characters (its typical use case), rather than removing them. This kind of exploration is also helpful if we need to separate truth from bits of incorrect information that Copilot may provide. This is why we need a baseline of our own Python knowledge!

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. Examples are shown here:

```
>>> first = 'This is a '
>>> second = 'sentence.'
>>> sentence = first + second      #1
>>> print(sentence)
This is a sentence.
>>> print('-=' * 5)     #2
-=-=-=-=-=
```

#1 Combines first and second strings and assigns result to sentence
#2 Repeats the -= string five times

4.3.5 #5. Lists

A string is great when we have a sequence of characters, such as 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:

```
>>> books = ['The Invasion', 'The Encounter', 'The Message']      #1
>>> books
['The Invasion', 'The Encounter', 'The Message']
>>> books.append('The Predator')                                  #2
>>> books
['The Invasion', 'The Encounter', 'The Message', 'The Predator']
>>> books.reverse()                                                #3
>>> books
['The Predator', 'The Message', 'The Encounter', 'The Invasion']
```

#1 A list with three string values in it
#2 Adds a new string value to the end of the list
#3 Reverses the list (now the values are in the opposite order)

Many Python types, including strings and lists, allow you to work with particular values using an index. You need to use indices whenever you want to work with part of a string or list rather than the whole thing. An index is just a number that identifies an element. Indices start at 0 for the first element and go up to, but not including, the number of values. The first value has index 0 (not index 1!), the second has index 1, the third has index 2, and so on. The last value in the list is at the index, which is 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 provides 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.
figure

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

Let's practice indexing on the current books list:

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

#1 books[0] corresponds to the first element.
#2 Error because index 3 is the last book!
#3 books[-1] refers to the last element in the list.

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]                    #1
['The Message', 'The Encounter']
```

#1 Starts at index 1, ends at index 2 (not 3!)

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) isn't 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]                                           #1
['The Predator', 'The Message', 'The Encounter']
>>> books[1:]                                           #2
['The Message', 'The Encounter', 'The Invasion']
```

#1 Same as using books[0:3]
#2 Same as using books[1:4]

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'      #1
>>> books[0]
'The Android'
>>> books[1] = books[1].upper()      #2
>>> books[1]
'THE MESSAGE'
>>> books
['The Android', 'THE MESSAGE', 'The Encounter', 'The Invasion']
```

#1 Changes books[0] to refer to the string value "The Android"
#2 Changes books[1] to be in all uppercase

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

```
>>> title = 'The Invasion'
>>> title[0]              #1
'T'
>>> title[1]
'h'
>>> title[-1]
'n'
>>> title[0] = 't'                   #2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

#1 Looking up a char works fine.
#2 But assigning doesn't!

A string is known as an immutable value, which means that you can't 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. If you get errors about a type not supporting item assignment, you're likely trying to change a value that can't be changed.

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 provides a summary of the features we covered in this chapter.

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 keywords 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.
Lists	`list = ['Leo', 'Dan']`	A sequence of values of any type. There are many powerful methods available for modifying lists.

4.4 Exercises

1. Recall the conditionals code we looked at in listing 4.2 (section 4.3.3). How does the behavior of this program differ from the original?

```
def larger(num1, num2):
    if num1 < num2:
        return num1
```

```
    else:
        return num2
```

2. In the following code, what gets printed if age is 32? What would be printed if you change every `if`, except for the first, to an `elif` instead?

```
if age >= 30 and age <= 39:
    print("Binging Friends")
if age >= 25 and age <= 34:
    print("Binging The Office")
if age >= 15 and age <= 24:
    print("Binging Pretty Little Liars")
if age >= 0 and age <= 14:
    print("Binging Chi's Sweet Home")
else:
    print("What's binging?")
```

3. What Python data type would you use to hold a series of temperature readings from a sensor that records data every minute? What would this data type store?
4. Arrange the following blocks of Python code to create a program that calculates the sum of all even numbers in a provided list.

```
for number in numbers:
total += number
if number % 2 == 0:
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
total = 0
print(f"The sum of all even numbers is: {total}")
```

5. (Recall section 4.3.1.) Consider the Python function call `min(input("Enter a number: "), input("Enter another number: "))`. Before running this code, think about what this line is supposed to do, and then answer the following questions:

   a. What do you expect to happen when this code is run? Describe each step of the execution process.

b. *How does Python handle the inputs and the* `min` *function together in this scenario?*
c. *If the inputs aren't numbers but strings that are numeric (e.g.,* `5` *or* `2`*), how might Python's behavior change, if at all?*
d. *What would be the outcome if one or both inputs aren't numeric? Test your hypothesis by trying different inputs.*

6. *In section 4.3.2, we talked about how variables persist in Python. How does this property of variables influence the way you might structure your code? Think of an example scenario where maintaining variable states across different parts of a program would be beneficial.*
7. *We ask Copilot to explain how the following code works:*

```
age = int(input("Enter your age: "))

if age < 13:
    print("You are a child.")
elif age < 20:
    print("You are a teenager.")
elif age < 60:
    print("You are an adult.")
else:
    print("You are a senior.")
```

Copilot provides the following explanation:

This Python code is a simple program that asks the user for their age and then prints out a message based on the age range they fall into.

The first line of the code uses the `input()` *function to prompt the user to enter their age. The* `input()` *function reads a line from input (usually from the user's keyboard), converts it into a string, and returns that string. In this case, the prompt is "Enter your age: ". The returned string is then converted into an integer using the* `int()` *function and stored in the variable age.*

The rest of the code is a series of conditional statements that check the value of age and print out a corresponding message. The `if` *statement checks if the age is less than 13, and if so, it prints "You are a child." If the age is not less than 13, the program moves to the next condition, the* `elif` *(short for* `else if`*) statement, which checks if the age is less than 20. If so, it prints "You are a teenager." If the age is not less than 20, it moves to the next* `elif` *statement, which checks if the age is less than 60. If so, it prints "You are an adult."*

It turns out that the explanation from Copilot is incomplete as it doesn't describe the code for all inputs. Determine what the explanation is missing, and add your answer to the explanation.

8. *Review the Python code provided and the initial prompt to Copilot. The explanation returned was too vague and didn't help you understand the string methods used in the code.*

```
text = "Hello, World! Programming is fun."
index = text.find("World")
new_text = text[index:].replace("World", "Everyone")

print(new_text)
```

Prompt given to Copilot:

Can you explain what this code does?

Copilot explanation given back:

The code involves manipulating a string stored in text by finding a specific word and replacing it with another word. It uses methods to locate where the word starts and then constructs a new string where the word has been replaced. Finally, it prints out the modified string.

Your task is to do the following:

a. *Identify why the initial prompt might lead to a vague or unhelpful explanation.*
b. *Improve the prompt to ensure a more detailed and specific explanation from Copilot. Consider what you specifically don't understand about the string methods and how the code uses them.*
c. *Write the revised prompt to Copilot, and discuss how your changes will lead to a better explanation.*

Summary

- *We need to be able to read code to determine whether it's correct, test it effectively, and write our own code when needed.*
- *Copilot can provide line-by-line explanations of code to explain what the code is doing, and you can use your own prompt to influence the type of explanation you get.*
- *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 (e.g., 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).*