2 First steps with AI-assisted coding

This chapter covers

- The fundamentals of GitHub Copilot
- Creating a Python environment for a Python project
- Building a project using generated code
- Using code completion to generate snippets
- Using the Copilot Chat interface

Getting started with generative AI tools can be daunting. Many people frequently ask me, “How do I get started? Which model do I choose?” Fortunately, the easiest way to start is to pick a tool and experiment. Once you do, you’ll find the learning curve surprisingly small.

We begin by using GitHub Copilot. To learn the basics, we’ll jump in headfirst to solve a programming problem. Next, we’ll build a useful application in Python with assistance from Copilot. By the end of this chapter, you’ll be comfortable using Copilot when writing code.

2.1 What is GitHub Copilot?

Do you remember the "rubber duck" debugging method? You visualize a rubber duck on your desk and explain the problem to it. You can use a real rubber duck or, like me, a toy wizard. I've confided in that wizard many times, working through problems. The key is to verbalize the matter to an imaginary helper. Once you articulate the problem clearly, by shifting focus, you might find solutions you didn't see before.

Imagine having an intelligent assistant that helps you with coding, like the rubber duck method, but one that can respond and suggest solutions. The tool will suggest word completions as you type, help you find and fix mistakes, and even make whole functions when asked. This is GitHub Copilot.

Launched in October 2021, GitHub Copilot is an AI coding tool powered by OpenAI's Codex algorithm. It employs extensive GitHub training data to offer context-aware code suggestions in multiple programming languages. It's an incredible tool that will surprise you with its usefulness. Copilot aims to boost your productivity and cut down on repetitive coding tasks. It's like having your own pair programmer or intelligent rubber duck providing suggestions as you work.

At the time of this writing, GitHub Copilot is available for Visual Studio, Visual Studio Code, Vim, NeoVim, JetBrains IDEs, and Azure Data Studio.

NOTE GitHub Copilot is not a free tool. There is a monthly fee for this tool. You can find more information on the GitHub Copilot information page (https://github.com/features/copilot). It is free for verified students, teachers, and maintainers of popular open source projects.

GitHub Copilot is trained on public source code using the OpenAI Codex model. Training involves providing a model with a large dataset of text and code to enable it to learn patterns and generate similar content. The model then uses machine learning to teach itself how to generate text and code that is like what it was trained on. GitHub's model was trained on billions of lines of code from GitHub repositories and other publicly available code. The Copilot service connects to and utilizes this model.

When you write code in the editor, it is sent to the Copilot service. The *context* is the code in your project that Copilot can view. The Copilot uses this context to gather information for queries. The code is sent to the model, and suggestions are returned to your editor. Then, when you accept the changes, this feedback is sent to the Copilot service. The model then uses this feedback to determine whether you liked what was sent, which forms a loop that gradually improves the suggestions over time. Figure 2.1 shows an overview of this process.

2.1.1 How GitHub Copilot works

Figure 2.1 is a good high-level example, but let's dig deeper into the process of how Copilot creates results. We can step through the entire process and understand how queries turn into code suggestions.

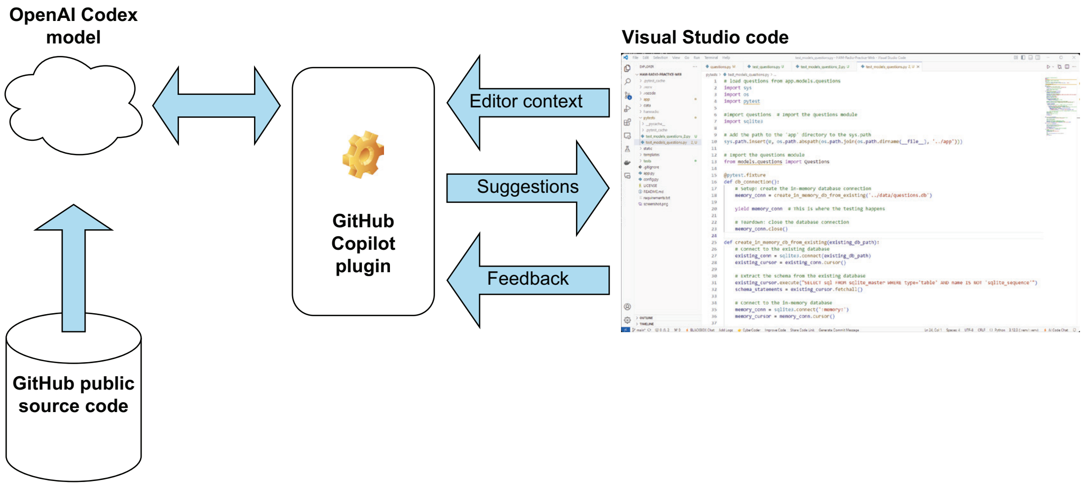

Figure 2.1 GitHub Copilot takes the code from your editor and analyzes it. It then sends suggestions back from the Codex model. Your decision to accept those suggestions or reject them is fed back into the model to refine the future suggestions.

INBOUND FLOW

Here's the process your prompt goes through as it's sent to GitHub Copilot. Understanding this process helps you generate better results by taking better control of the tool.

The process begins in your code editor, either as a comment or chat message. The prompt itself is sent with important context, such as

- Code before and after the cursor
- Code and information in open tabs
- Filename and file type
- Project structure and paths
- Programming language and frameworks used

Fill-in-the-middle (FIM) refers to a technique where Copilot analyzes both the code preceding the segment selected or asked about and the segment that follows it. FIM allows Copilot to understand a greater amount of code.

This code is then sent to a proxy server, which acts as a filter. This filter prevents attempts to hack the LLM or manipulate the system in a harmful way. The next step is a toxicity filter, which looks for hate speech and inappropriate content. Anything that could be harmful or offensive is blocked at this point. It also filters out personal information such as names, addresses, or identifiable information.

Finally, the prompt is prepared and shipped to the LLM model to generate code suggestions. These suggestions are based on the prompt, but also on the surrounding context. The model then uses all this information to create the best possible answer. This process happens in seconds and is illustrated in figure 2.2.

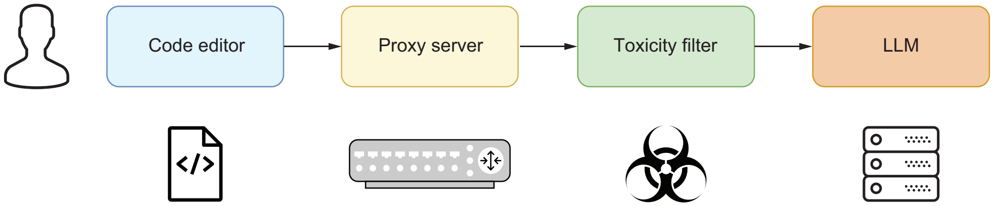

Figure 2.2 The inbound flow to GitHub Copilot

Outbound flow

The outbound flow is in the reverse direction. First, the result is passed through a toxicity filter to ensure no hate speech, inappropriate, or private material is sent from the LLM. Then, the proxy server runs a final check for

- *Code quality*—Checks for common bugs and bad patterns.
- *Security*—Checks for known vulnerabilities and insecure patterns.
- *Matching public code* (optional)—Checks whether the content is original and how closely it matches public code. This code can then be removed or truncated if the Copilot administrator wants to do so.

This process is illustrated in figure 2.3, as the suggestions travel back to the IDE.

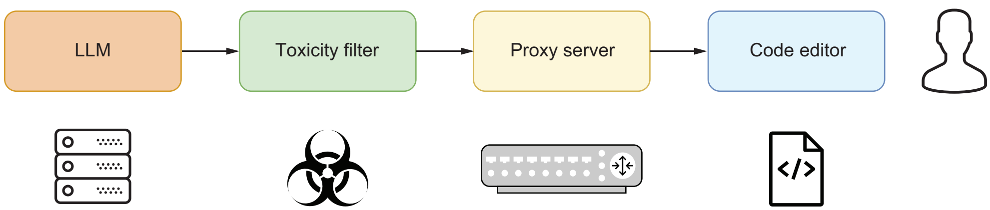

Figure 2.3 Outbound flow from the LLM back to the code editor

This process is ongoing as you use the GitHub Copilot application in your IDE. This continuous loop ensures your code suggestions are appropriate and useful for your project. This is one of the key distinctions between GitHub Copilot and ChatGPT when handling code suggestions. The context and filtering can make a big difference in the quality of work.

2.1.2 Interacting with GitHub Copilot

There are several ways to interact with Copilot. You can work from the editor with assistance or open a separate Copilot chat window. Here are the ways Copilot can work for you as a developer.

Code completion

Copilot can suggest code completions as you type in the editor. It analyzes the current context of your code and provides real-time suggestions to complete statements or functions. This can speed up coding and reduce syntax errors (figure 2.4).

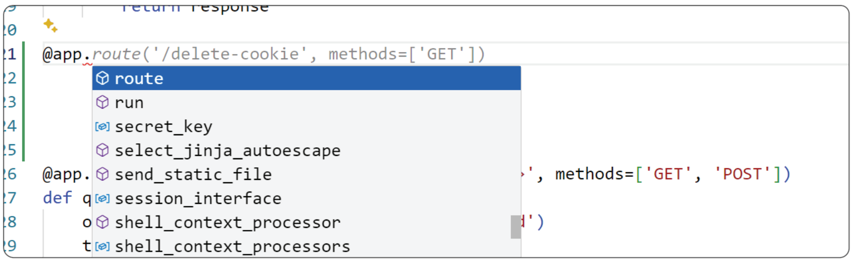

Figure 2.4 Copilot can suggest code completions as you type by analyzing comments or surrounding code for context.

In the previous example, we were building an API. GitHub Copilot knows a lot about APIs and how to create them. You can describe API endpoints and structure, and Copilot will generate appropriate data classes and request handling code automatically, saving you time.

Code generation

Copilot can generate full segments of code based on statements, comments, or partial code snippets. You can enter a comment describing a function, and Copilot will attempt to complete the code. This feature is great for repetitive tasks and when looking for different ways to solve a problem (figure 2.5).

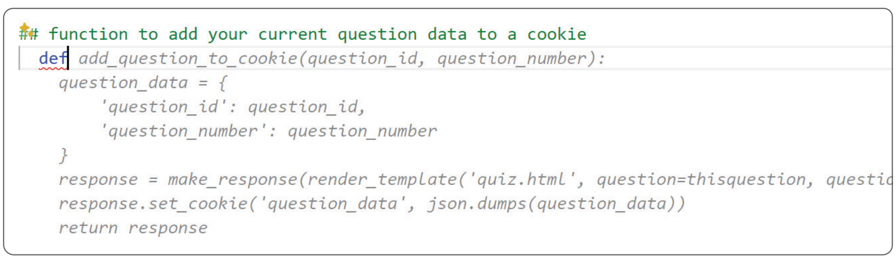

Figure 2.5 Copilot can generate entire functions or segments of code based on comments or partial code snippets.

Translation

You can translate code from one language to another by highlighting the code and invoking Copilot. This feature allows you to convert legacy codebases into newer or modern languages. It can also help you understand what a function is doing by translating it into a language you're more familiar with (figure 2.6).

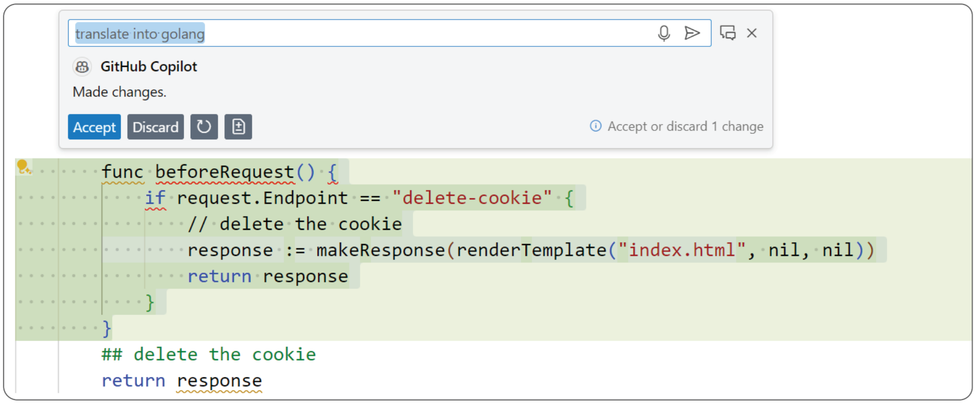

Figure 2.6 You can use Copilot to translate code from one language to another. Here, I have highlighted a section of Python code and asked Copilot to translate it into Go.

Debugging

You can highlight a problematic piece of code and ask Copilot to debug it. It can provide suggestions to fix syntax errors, logical errors, or performance problems, which lets you quickly identify and resolve bugs and speed up your debugging process (figure 2.7).

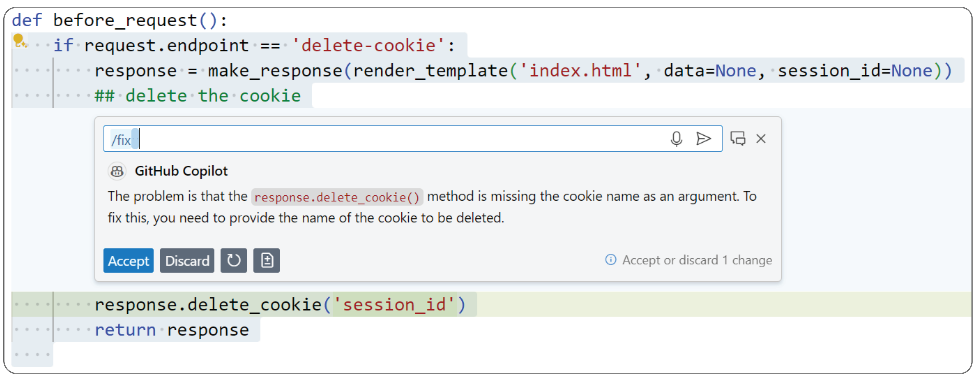

Figure 2.7 You can highlight a problematic section of code and get suggestions on how to fix it. You can either accept the answer and have the code inserted or discard it and attempt a fix of your own.

Documentation generation

You can ask Copilot to generate documentation for you. This is handled either in the editor window or through the chat window. You can use the chat window to have conversations with Copilot and ask questions or get ideas (figure 2.8).

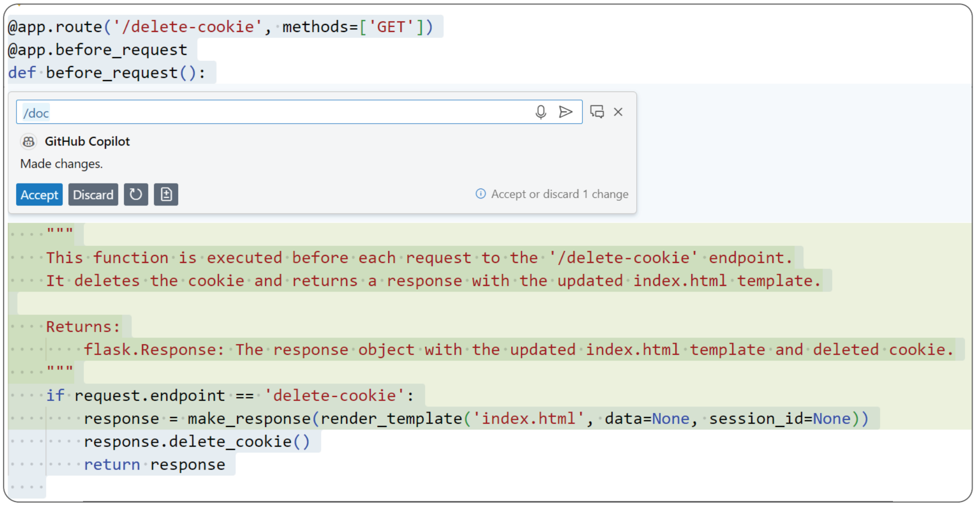

Figure 2.8 You can automatically generate comments and other code documentation using Copilot. This is one of my favorite features of the tool, as it saves a tremendous amount of time.

Right-click tool windows

It is possible to interact with Copilot in many ways inside your IDE window. There are some shortcuts for the features mentioned before. For example, you can click in the IDE near the code and ask it to document it, explain it, fix it, or generate tests, as shown in figure 2.9.

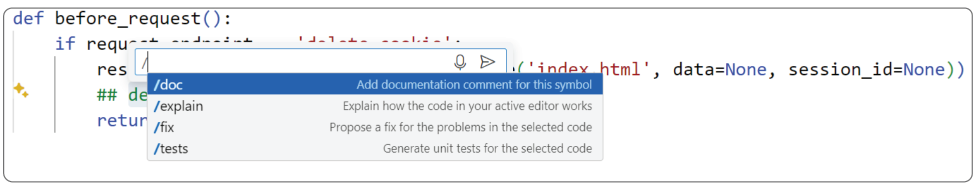

Figure 2.9 GitHub Copilot can perform many actions from a right-click menu within your IDE.

Chat window

You can interact with GitHub Copilot in a chat window as well, and it has a familiar dialog interface like ChatGPT and other models. The difference is that this chat window is fully aware of the code in your project. In fact, some commands in the IDE will send you to a chat window so you can ask more questions or attempt to regenerate the results (figure 2.10).

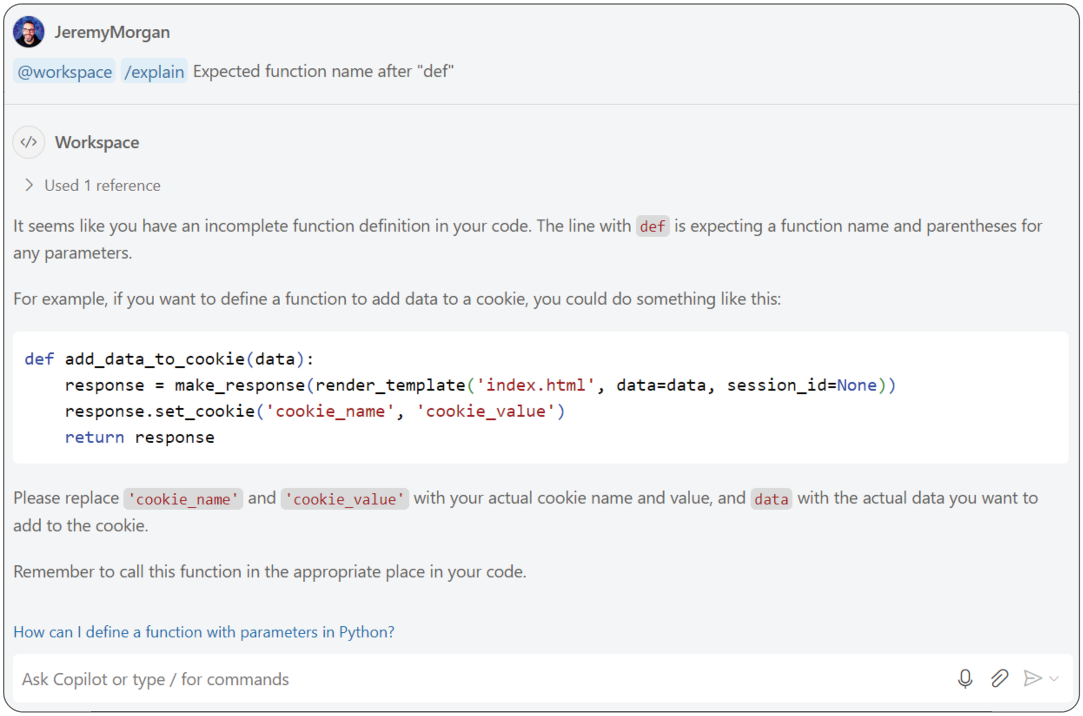

Figure 2.10 Copilot has a traditional chat window interface within Visual Studio, making it easy to ask for help or suggestions.

We're going to explore this tool through practical, hands-on activities. You'll learn how Copilot can assist in auto-generating Python code as we build an application to analyze text. We'll explore how Copilot understands context from comments and function names. With Copilot's help, we can put together a working script quickly.

> **TIP** You can use "hints" to guide Copilot to generate exactly what you're looking for. For instance, if you want an asynchronous function, you can type in "async def," and that will tell Copilot you want a coroutine instead of a normal function.

By the end of this chapter, you'll be ready to bring this AI coding sidekick into your own projects. Copilot may not yet write flawless code, but it can make you a more productive programmer. Let's dive in and see what this tool can do. We'll build an app that does some primitive natural language processing (NLP), which is appropriate since we use tools that utilize it.

2.2 Common patterns

As you continue working with generative AI coding tools, you'll develop patterns when working with chat windows. These patterns will become second nature to you after a while, and you can use this approach for many different tools. Here are some common phrases you can ask in chat to get better results:

- *Find the error in the selected code.* You highlight a code snipped and ask it to diagnose bugs or problems with syntax. This phrase works especially well with tools that keep your application in context and consider that context for the result.
- *Refactor this code to make it better.* While this can be highly subjective, there are times when Copilot and other tools can suggest a better pattern or algorithm for what you're trying to do. You can ask it to optimize for security, speed, or resource usage.
- *Generate code for {specific task}.* This is a common way to generate boilerplate code for things you've done countless times but don't want to write out by hand. Things such as connecting to databases, reading from files, and sending HTTP commands are great examples of code that can be generated this way.
- *Explain how this code works.* This pattern is invaluable when working with legacy code or anything you didn't build. It can quickly get you up to speed on what the code is designed to do, and all the tools we'll talk about do a great job at it.
- *Add tests for this code.* This pattern can sometimes feel like cheating. The tools we discuss are great at generating tests. This feature gives you a clear time advantage, as it generates tests for you rather than writing them out by hand. It provides a quality advantage as well, because it can find cases you haven't considered or forgot to add. This is a valuable pattern you'll use repeatedly.
- *Suggest improvements for readability and maintainability.* This is another helpful pattern for refactoring or working with legacy code. Admit it, all of us have put something together just to make it pass tests and fix a problem. It wasn't the clearest or simplest way to do it, and this is a way to root out code like that and improve it.
- *Compare tradeoffs for {option a} and {option b}.* Sometimes, our first idea isn't the best approach. Sometimes we're tired, burned out, or just can't think of the best way to do something. This is where our AI powered assistants can really shine by generating different ways to approach a problem and weighing out the best choice.

2.3 Context is everything

Coding assistants seem to just know what you're trying to do, and it's not magic. Tools such as GitHub Copilot, Tabnine, and BlackboxAI use context to generate smarter, more relevant suggestions. They don't just analyze a single line of code, but they pull in information from the entire workspace, the files you've got open, and even the structure of your project. This means that when you're writing a function, Copilot isn't just guessing—it's making an educated suggestion based on your existing code. Generative tools in IDE integration draw insights by analyzing

- Code from the current file being edited
- Content from neighboring tabs in the IDE
- The entire workspace or solution structure
- Code before and after the cursor position (Github Copilot)

As for project understanding, the insights come from

- Repository-level context
- Pull requests and open problems
- Project documentation and configuration files

The level of product you're using makes a difference here. Often, the enterprise or professional level products give a deeper dive than individual licenses. However, all three IDE-based tools we'll use look at the file being edited and files in your project.

One of the biggest tricks up Copilot's sleeve is something called "fill-in-the-middle" (FIM). Instead of only looking at what you've typed so far, it scans the surrounding code before and after your cursor to generate more accurate completions.

This is a game-changer because it allows AI to suggest code that fits seamlessly within your existing logic. The result? Better suggestions for your project instead of generic solutions.

Now let's jump right into using GitHub Copilot for a project. We will build an application that uses NLP to analyze and count words in a document. But first, let's look at what NLP is. It's important for the project and for our understanding of generative AI tools.

2.4 What is NLP?

Have you ever wondered how virtual assistants such as Siri or Alexa understand what you're saying? It can seem like magic, but it's natural language processing, or NLP. NLP combines linguistics, computer science, and artificial intelligence to allow computers understand, interpret, and generate human language.

NLP systems aim to take text input, process it, interpret the words, and produce meaningful output. These applications perform sentiment analysis, machine translation, and text summarization. NLP isn't just for virtual assistants. It's also used in fields from customer service to healthcare by allowing computers to communicate with humans effectively.

A core concept of NLP is tokenization, which breaks down text into smaller units called tokens. These are usually words, punctuation, or phrases. Tokenization helps computers understand the structure and attempt to evaluate the meaning of text by breaking it into smaller chunks (figure 2.11).

The words and punctuation are broken into chunks, as shown in figure 2.11, and given an identifier. NLP systems look for patterns in words, such as

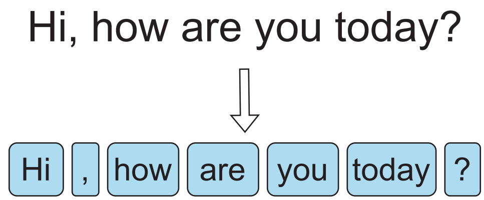

Figure 2.11 Word-based tokenization breaks whole words and punctuation into chunks that can be processed and analyzed.

- *Word frequency*—What words are used the most? Which are used the least? Is this an indicator of what the subject is about?
- *Word types*—NLP distinguishes between the types of words used: nouns, verbs, adjectives, and similar. This helps with assembling sentences and extracting meanings.
- *Word co-occurrence*—NLP analyzes which words are frequently found together in text. This can reveal semantic relationships between these words.
- *Text classification*—Words are analyzed and fit into predefined categories. They're then analyzed for tasks such as spam detection, document categorization, and sentiment analysis.

Word-based tokenization is core to NLP, but it's just one of the many ways NLP analyzes text. This is a high-level overview of NLP and not the focus of this book. However, this is what our next project is about, and NLP is used extensively with our tools, so if it interests you, it's a great subject to learn more about.

2.5 A simple Python project

As our first Copilot-assisted program, we will perform some crude NLP. We'll take a text file from Project Gutenberg and count the word frequency. Then, we will rank the words and see which ones were used the most. Finally, we'll build and view a bar chart of the results. All of this will be done with a simple Python script that we build, and we'll use GitHub Copilot as our smart assistant.

For this example, we'll use GitHub Copilot in Visual Studio Code. We'll use Copilot to create the code for our program based on small descriptions and a conversation with Copilot Chat. Copilot Chat is an extension for Visual Studio code with a chat-like interface. You can ask the tool questions, and it will produce answers.

In addition, we'll use a combination of Visual Studio Code, Python, and the GitHub Copilot extensions. If you're following along, you'll need all three of these installed on your machine.

For this project, we need a big text file with a lot of words. I selected a file from Project Gutenberg and analyzed it. The text is one of my favorite books on that site, *Free Air* by Sinclair Lewis. The full-text version of this novel is available at https://www.gutenberg.org/cache/epub/26732/pg26732.txt.

I want to know which words are used the most frequently in this novel. I will count all the words in this text file and rank them by frequency to learn this.

Next, I need to load up the text and parse it. I need to identify a "word" by anything enclosed with spaces. If there is a space before or after a set of characters, we assume it's a word. We must also prepare the text to ensure our word count is accurate. Here's an example.

The words "truck," "Truck," and "truck!" are considered three distinct words. The resulting data will be scattered everywhere if we look at how many times a truck is mentioned in different ways. We can solve this problem by making all the letters lowercase and removing punctuation. This way, data will show the word "truck" being mentioned three times, which is what we're looking for.

We need to tokenize the text. This is what we just learned about in the previous section, so let's put it into action. Consider the sentence "My name is Jeremy, and I like apples." After tokenization, it becomes

```
My
Name
is
Jeremy
,
and
I
like
apples
.
```

Each word and punctuation mark are treated as a separate token. The tokens become the building blocks the computer uses for text analysis and processing. By breaking the text into tokens, the computer can attempt to understand the content and meaning behind the words. The word itself and placement are important here. The process of splitting up text like this is called *tokenization.*

We will parse every token, count the unique tokens (words), and count the identical ones. We don't need to count punctuation as we're only concerned about word count here. Punctuation will provide irrelevant data. This will give us accurate data to find which words are used the most.

Let's review our primary objectives for this application:

- Open a text file and read it.
- Remove any punctuation and convert all text to lowercase.
- Tokenize the text into individual words.
- Count the frequency of each unique word as quickly as possible.
- Display the top N most frequent words in a bar chart.

Figure 2.12 shows what the flow looks like.

This first application is simple enough we can build it here. Let's do it!

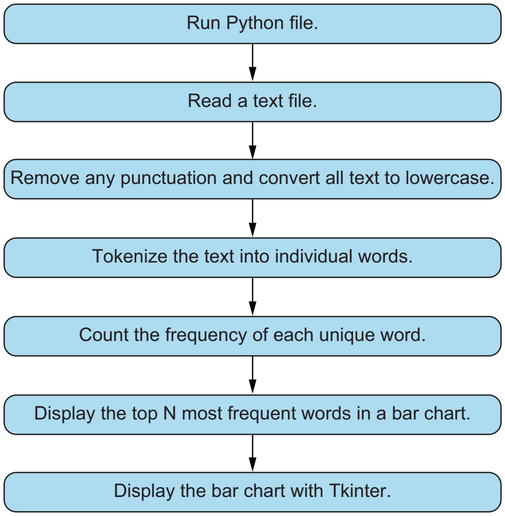

Figure 2.12 The workflow of our sample application

2.5.1 *Preparing your development environment*

Let's familiarize ourselves with our environment. First, you'll want to create a folder for the application to live in. Next, open that folder with Visual Code going to File and then Open Folder. You'll see a blank VS Code project with the welcome page (figure 2.13).

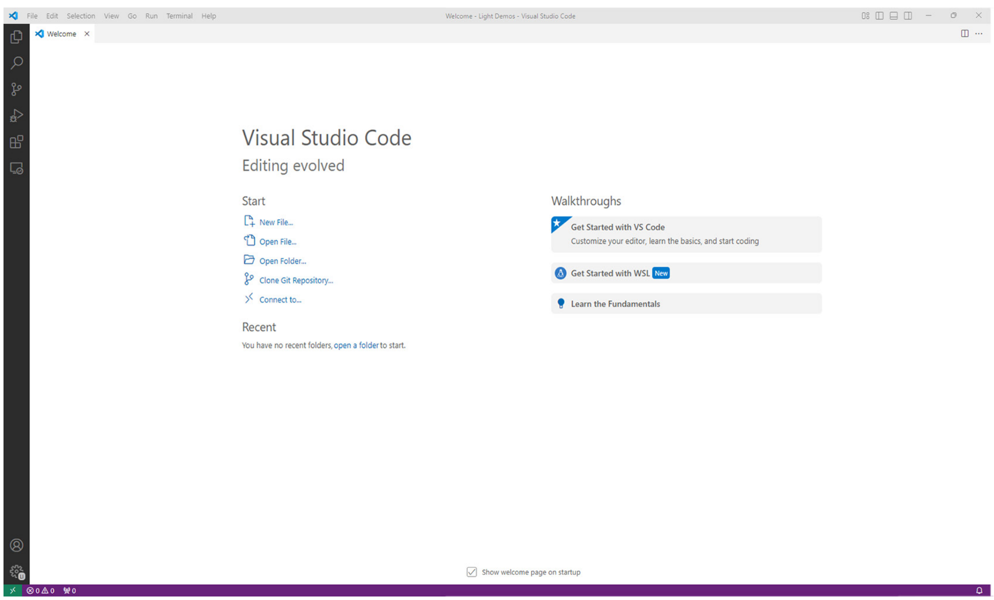

Figure 2.13 Visual Studio Code opened to a new folder

Next, we want to enable the terminal. You have several different options here, depending on your operating system. In Windows, among the rest, you can enable

- A PowerShell console
- Windows Terminal
- WSL (Windows Subsystem for Linux) Terminal

On a Mac or in Linux, you can enable the console, which will use a default OSX or Linux console.

Go to Terminal and then to New Terminal from the main menu to create a new terminal. You'll find it in the lower right corner. I generally keep this window open while developing. Figure 2.14 shows the "terminal" I'll be referencing when I suggest "entering something into the terminal."

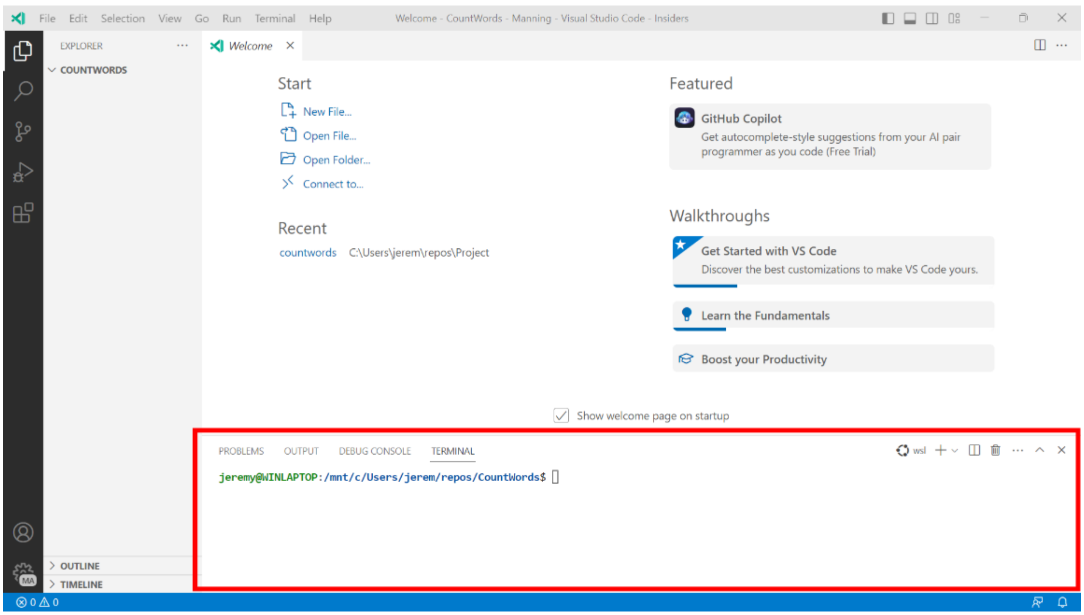

Figure 2.14 Location of the Terminal window

I prefer to use WSL in Windows and open that in the terminal window in Visual Studio since I spend most of my time in Linux. But if it must be in Windows, WSL makes everything consistent at the command line. If you're comfortable in Windows, you can use a PowerShell or Command Prompt.

Learning to navigate and use the terminal effectively is essential, regardless of the operating system you select. If you're already a Python developer, you're likely already familiar with the terminal. It makes creating and moving files around easy, while working with Python scripts. Figure 2.15 shows the layout of our Visual Studio Code Desktop.

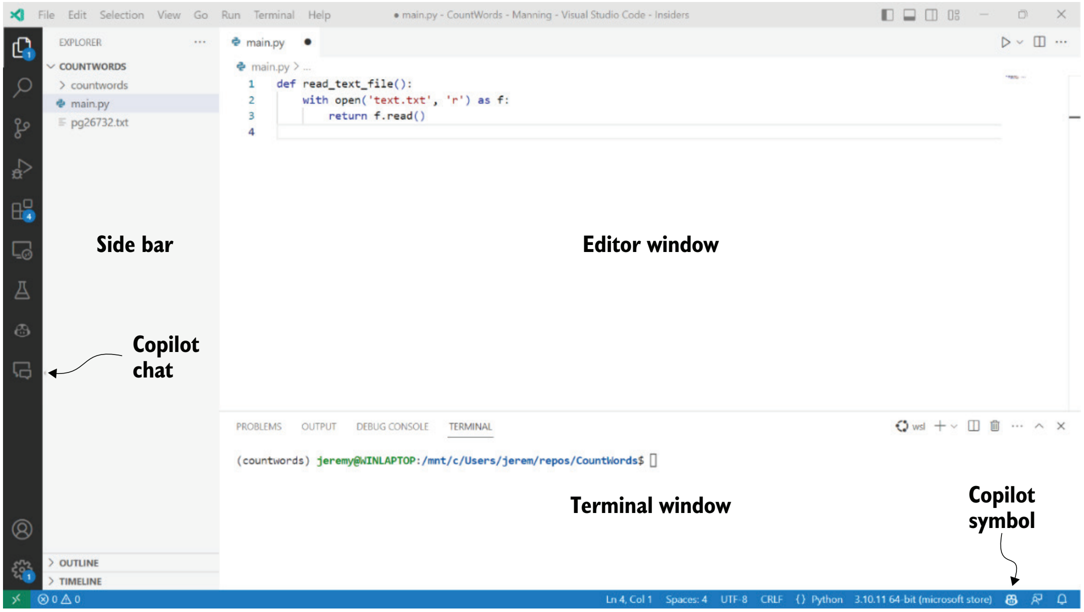

Figure 2.15 The Visual Studio Layout, showing Copilot is accessible in the activity bar, and the lower taskbar

I'll use the following terminology moving forward. These are the basic components of Visual Studio code:

- The *side bar* lists files so you can easily navigate between them.
- *Copilot Chat* is located to the left of the sidebar in the activity bar.
- The *terminal window* is where we'll enter terminal commands.
- The *editor window* is where we'll enter code.

We will utilize Copilot to generate and analyze code for us. As discussed earlier, you can interact with Copilot in the following ways:

- *Adding a comment to your code*—You can add a comment describing something you want to happen, and Copilot will attempt to generate a solution.
- *Code completion*—You start writing code, and Copilot will attempt to complete the statement or method you're trying to build.
- *Copilot Chat*—You can interact with Copilot Chat like a person. Ask questions, debug snippets, and more.

2.5.2 Creating the application

We will use a Python virtual environment for this application. Regardless of the project size, I create virtual environments for it. This is essential for maintaining clean and organized environments for Python development. It creates an isolated environment for the project, so you can ensure each project has its own versions of packages, libraries, and Python. This prevents conflicts between packages and avoids large messes that can arise from conflicting package versions. Virtual environments take seconds to create but can save you hours of work in the long run.

First, let's create a folder for our application. In that folder, we will create the Python virtual environment. Open the terminal window, and at the prompt, type

```
python -m venv countwords
```

Then, activate the environment:

```
source countwords/bin/activate
```

You should see an indicator before your prompt with the name of the environment in the terminal (figure 2.16):

Figure 2.16 A terminal window showing an active Python environment

I recommend using a Python virtual environment for all your projects. It helps decouple your Python installations. This approach makes your projects more portable and less reliant on globally available packages, which you may want to change. If you have multiple projects with multiple libraries, it's the best way to work. It's quick to set up, so it's worth getting into the habit of always creating them.

For instructions on how to create virtual environments, including in Windows, check out the guide at https://code.visualstudio.com/docs/python/environments.

Now that I have the environment created, I want the text file of *Free Air* to live on my hard drive. You can copy this text and save it to your drive, download it from the browser, or use a utility such as `wget` to download it:

```
wget https://www.gutenberg.org/cache/epub/26732/pg26732.txt
```

This utility is available on all three platforms (Windows, Linux, and Mac), so I highly recommend installing it. It will download `pg26732.txt` to your hard drive. So now that we have our text file, let's build our app.

Step 1: Reading a text file

Our first objective is to open our text file so we can read it. Let's create a `main.py` file and create our first function. We'll use the code completion method to interact with Copilot here. This happens when you type in some code, and the Copilot attempts to autocomplete it for you.

As we type in the line

```
def read_text_file(file_path):
```

Copilot has a suggestion for us immediately (figure 2.17).

Figure 2.17 A typical Copilot suggestion

Copilot assesses what I'm trying to do based on what I've named the function. It gives the following two options:

Option 1:

```
def read_text_file(file_path):
    with open(file_path, 'r') as f:
        return f.read()
```

Option 2:

```
def read_text_file(file_path):
    """Reads the text file and returns the text as a string"""
    with open(file_path, 'r') as file:
        text = file.read()
    return text
```

The two options are functionally identical. Both read a text file and return the text as a string. However, option 2 is more readable. It contains a docstring and uses `file` for the variable name instead of `f`. It also returns text as a variable instead of simply returning the output of the `read()` function.

But let's take a step back and do a little test. Let's write the function *inefficiently* and see how Copilot handles that. To do this, we'll use the Copilot Chat extension. Note that at the time of this writing, Chat is in beta, so it will likely look a little different (see figure 2.18).

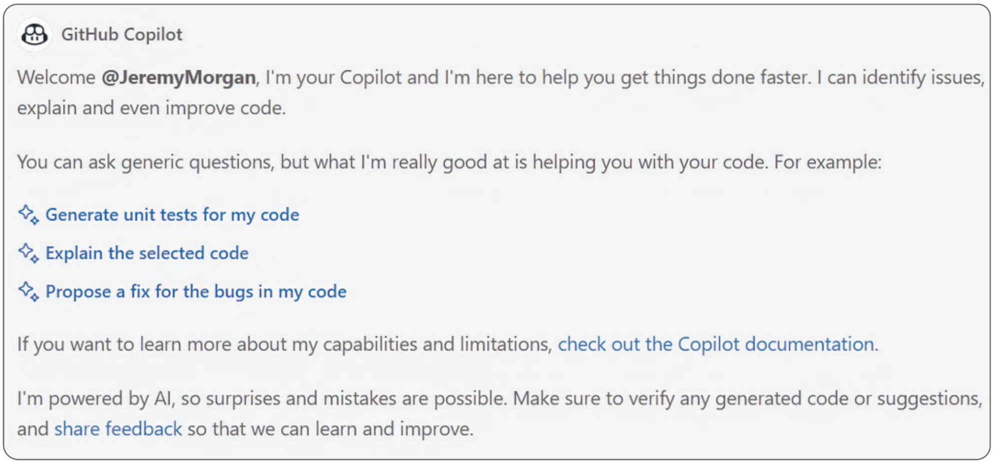

Figure 2.18 The GitHub Copilot Chat welcome screen

I have written some code that's not so great. It adds a list to the operation that isn't needed and doesn't automatically close the file. It's not horrible, but it can be improved:

```
def read_text_file(file_path):
    file = open(file_path, 'r')
    lines = []
    for line in file:
        lines.append(line)
    file.close()
    text = "".join(lines)
    return text
```

Let's ask GitHub Copilot Chat how it would improve this code. I open Copilot Chat and type in a question. I use a simple, plain-English prompt. With that prompt in the chat window, I paste the code to let Copilot know which code I'm referring to.

JM How can I improve this code?

It's worth noting that my prompt was very simple. This is all you need for most cases; however, you can implement prompt engineering for more complex problems. Generally, you won't need to assign a role as you do with other tools, as Copilot is already set up for this. I press Enter to send my prompt (figure 2.19). As shown in figure 2.20, Copilot outputs a response with a cleaner, more efficient method.

These are two ways we can use Copilot to create this function. We can either name the function cleverly and get code automatically suggested, or write code ourselves and ask GitHub to refactor it.

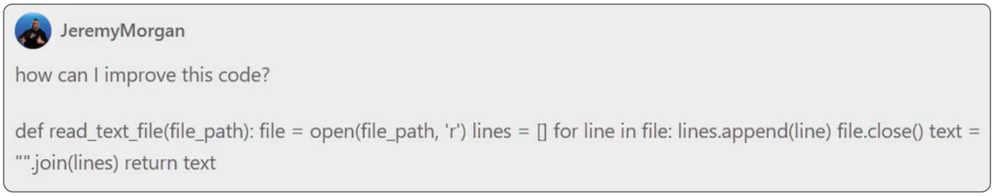

Figure 2.19 Prompt entered in Copilot Chat

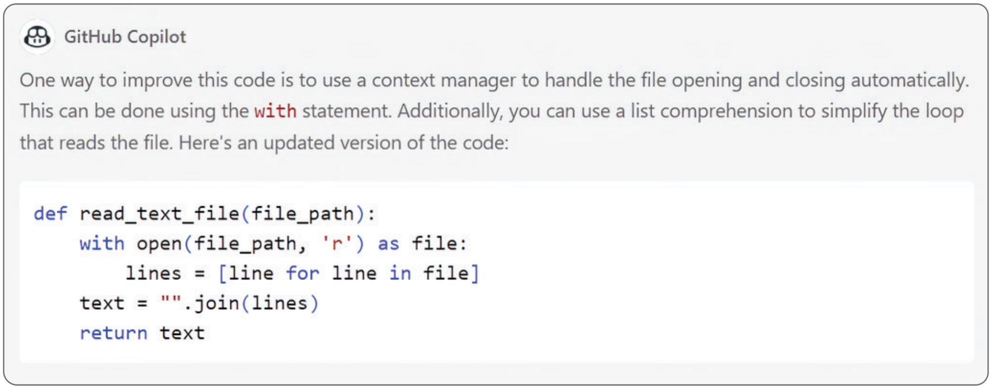

Figure 2.20 A response from Copilot Chat

Copilot's code generation capabilities are neat, but we don't want to rely on them too heavily. Don't assume this is optimized or that it's been tested or validated. Always evaluate the generated code and make sure you understand it.

In our case, let's take this code snippet from the Copilot Chat window and paste it into our editor window (which is the source code for our application):

```
def read_text_file(file_path):
    """Reads the text file and returns the text as a string"""
    with open(file_path, 'r') as file:
        text = file.read()
    return text
```

Note that in this generated code, there is no encoding specified. This is an example of Copilot generating code that works but may not be optimal. The code runs fine on my Linux computer, but if you open it in Windows, you may have a problem. You can change the open function to include encoding if you want to make it safer to use. Try something like

```
with open(file_path, 'r', encoding='utf8') as file:
```

Occasionally, manual intervention like this is needed with generated code. Make sure to always review the code that is output.

Step 2: Remove any punctuation and convert all text to lowercase

For this step, we'll do something a little different. Let's try generating this function from a comment in the code. I created the following comment:

JM # create a function to convert the text to lowercase and remove punctuation.

We're given three different options, as indicated in the toolbar that pops up (figure 2.21).

Figure 2.21 The options are shown in the Copilot toolbar.

You can switch between these options to see examples of code generated by Copilot.

Note that even if you type in this exact prompt with this exact setup, you likely won't get three suggestions. You may see anywhere from one to five. They most likely won't be the same as this example. Many factors affect what Copilot generates, and it's dynamic. This is the nature of generative AI.

> **Why is the output different every time?**
>
> Most large language models (LLMs), including GitHub Copilot, are deterministic in operation, but the post-processing actions produce varied results. This means they can generate different outputs each time. Even under similar conditions, your output may differ with the same input. This variability is both a natural feature and a benefit of these models, allowing for more creative and diverse responses.

Let's examine the options given.

Option 1

```
def clean_text(text):
    """Converts the text to lowercase and removes punctuation"""
    text = text.lower()
    punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
    for character in punctuation:
        text = text.replace(character, '')
    return text
```

This is a decent option, and I like it. It's clean and easy to understand what's happening. We have some big text replacement operations happening here that can take some time. We're parsing the entire document 30+ times, so that will add some time. However, this is something we'd reasonably expect. Let's look at option 2:

Option 2

```
def clean_text(text):
    """Converts the text to lowercase and removes punctuation"""
    text = text.lower()
    punctuation = '.,;:"!@#$%^&*()-_+=~`[]{}|\/<>?'
    for mark in punctuation:
        text = text.replace(mark, '')
    return text
```

This option isn't significantly different. Variable names are changed, and there are fewer punctuation marks. Finally, option 3 is

Option 3

```
def clean_text(text):
  """Converts the text to lowercase and removes punctuation"""
  # convert to lowercase
  text = text.lower()
  # remove punctuation
  punctuation = ['.', ',', ';', ':', "'", '"', '!', '?', '-', '(', ')']
  for punc in punctuation:
      text = text.replace(punch, '')
  return text
```

This version is interesting because it treats each punctuation mark as individual items in a list instead of a string, which I suspect will be faster. Let's find out if that's true.

2.5.3 Side quest: Testing the function speed

Since this is a significant part of our application, we want to find out which of these functions are faster, if any. This is a great way to check the code that Copilot is producing. We're doing this for two reasons:

- We want to know which function is more performant.
- We want to review and check the code Copilot is giving us.

Remember that Copilot isn't producing perfect code that's ready to use. You need to check the code it outputs every time. Testing, profiling, and debugging are still needed. Having knowledge about the languages enough to smell when something isn't right is crucial—trust but verify. Let's verify which method is the most optimal from a performance standpoint as an example.

We'll break it into two options based on how the punctuation data is stored. We'll test options 1 and 3.

In our file, we create the following two functions:

```
def clean_text1(text):
    """Converts the text to lowercase and removes punctuation"""
    text = text.lower()
```

```
    punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
    for character in punctuation:
        text = text.replace(character, '')
    return text
def clean_text2(text):
  """Converts the text to lowercase and removes punctuation"""
  # convert to lowercase
  text = text.lower()
  # remove punctuation
  punctuation = ['.', ',', ';', ':', "'", '"', '!', '?', '-', '(', ')']
  for punc in punctuation:
      text = text.replace(punc, '')
  return text
```

And we'll create the following code the functions to time them. I asked Copilot Chat:

JM How can I write code to call cleantext1 and cleantext2 on the text from read_text_file and time them to see which is faster?

It generated the following code, which looks like something I would have created.

```
text = read_text_file('pg26732.txt')

start_time = time.time()
clean_text1(text)
end_time = time.time()
print(f"clean_text1 took {end_time - start_time} seconds")

start_time = time.time()
clean_text2(text)
end_time = time.time()
print(f"clean_text2 took {end_time - start_time} seconds")
```

Awesome. So, I run it a few times (figure 2.22):

```
(countwords) jeremy@KATANA:/mnt/c/Users/jerem/Projects/CountWords$ python speedtest.py
clean_text1 took 0.005719661712646484 seconds
clean_text2 took 0.0026023387908935547 seconds
(countwords) jeremy@KATANA:/mnt/c/Users/jerem/Projects/CountWords$ python speedtest.py
clean_text1 took 0.005422830581665039 seconds
clean_text2 took 0.0026755332946777344 seconds
(countwords) jeremy@KATANA:/mnt/c/Users/jerem/Projects/CountWords$ python speedtest.py
clean_text1 took 0.0050449371337890625 seconds
clean_text2 took 0.0026094913482666016 seconds
(countwords) jeremy@KATANA:/mnt/c/Users/jerem/Projects/CountWords$ python speedtest.py
clean_text1 took 0.00489354133605957 seconds
clean_text2 took 0.002547025680541992 seconds
(countwords) jeremy@KATANA:/mnt/c/Users/jerem/Projects/CountWords$ python speedtest.py
clean_text1 took 0.0053517818450927734 seconds
clean_text2 took 0.002520322799682617 seconds
(countwords) jeremy@KATANA:/mnt/c/Users/jerem/Projects/CountWords$ python speedtest.py
clean_text1 took 0.005056858062744141 seconds
clean_text2 took 0.002579927444458008 seconds
(countwords) jeremy@KATANA:/mnt/c/Users/jerem/Projects/CountWords$
```

Figure 2.22 Output from our program

Neither one is slow, but it appears `clean_text2` is the faster function here. We'll use that. I've added additional punctuation marks by asking Copilot Chat to generate a list of additional punctuation marks.

JM # Please create a thorough list of punctuation marks to avoid, including brackets, braces, and slashes

The final function is

```
def clean_text(text):
  """Converts the text to lowercase and removes punctuation"""
  # convert to lowercase
  text = text.lower()
  # remove punctuation
  punctuation = ['.', ',', ';', ':', "'", '"', '!', '?', '-', '(', ')', '[',
     ']', '{', '}', '/', '\\', '|', '<', '>', '@', '#', '$', '%', '^', '&',
     '*', '_', '+', '=', '`', '~']
  # clean the text
  for punc in punctuation:
      text = text.replace(punc, '')
  return text
```

Now, we can move on with our application.

Step 3: Tokenize the text into individual words

Next, we need to count individual words. This is simple to do in Python, and we can add this to our clean text function. Creating an entire function that only splits text is unnecessary. I'll add a line above `return text` and type the following comment:

JM # tokenize the text into words

It generates the following line of code, and now we'll return `words[]` instead of text.

```
# tokenize the text into words
words = text.split()
return words
```

Step 4: Count the frequency of each unique word

Now, we want to count the frequency of each unique word. Let's try a comment prompt for that.

JM # count the frequency of each unique word in words

This prompt gives us a single option that looks fine. It could be improved by using `Defaultdict()` from the collections module. Again, Copilot usually generates code that works, but it may not be as good as code generated by an experienced developer. Your suggestion may also show something completely different from this example.

```
def count_words(words):
    """Counts the frequency of each unique word in words"""
    word_count = {}
    for word in words:
        if word in word_count:
            word_count[word] += 1
        else:
            word_count[word] = 1
    return word_count
```

Awesome. This program is coming along nicely. Now, we need to display our results in a human-readable format.

STEP 5: DISPLAY THE TOP N MOST FREQUENT WORDS IN A BAR CHART

Now for the fun stuff. Let's generate a function that builds a graph in Matplotlib. This tool produces clean and useful graphs with a few simple parameters and helps you display data in an interesting way. Here's the prompt we'll use:

JM # create a function to plot out the top n words on a graph with matplotlib

This gave me a few options, but here's the one I chose. It's clear and easy to understand and includes comments.

```
def plot_word_count(word_count, n):
    """Plots the top n words in a word_count dictionary"""
    import matplotlib.pyplot as plt
    # sort the word_count dictionary by value
    word_count = sorted(word_count.items(), key=lambda x:
➥x[1], reverse=True)
    # get the top n words
    top_words = word_count[:n]
    # create the plot
    plt.bar(range(n), [x[1] for x in top_words], align='center')
    plt.xticks(range(n), [x[0] for x in top_words])
    plt.xticks(rotation=70)
    plt.xlabel('Word')
    plt.ylabel('Frequency')
    plt.title('Top {} Words in the Text'.format(n))
    plt.show()
```

Now, we just need to tie these functions together. Can Copilot do this for us?

STEP 6: PUTTING IT ALL TOGETHER

Now it's time to put it all together into a main function. I want a single function that calls the functions we just built in the right order. I want this function to run first every time someone executes the Python file. Here's the comment prompt I used:

JM # create a main function that runs when the Python file is run and execute the functions

I wasn't sure this would work out, but it did. Here's the generated function:

```
def main():
    """Main method that runs when the python file is run"""
    # read the text file
    text = read_text_file('data/hamlet.txt')
    # clean the text
    words = clean_text(text)
    # count the words
    word_count = count_words(words)
    # plot the top n words
    plot_word_count(word_count, 20)
```

I need to rename the file it calls to the name of my text file. I'll change `'data/hamlet.txt'` to the filename I downloaded:

```
text = read_text_file('pg26732.txt')
```

Now, assuming we don't know Python, if you run this file, nothing will happen. I'll create a comment in the code above the method to prompt Copilot. I want it to generate code to make this our "main method" executed when the file is run by itself.

I will give the following comment prompt:

JM # execute the main method if this file is run

It generates

```
if __name__ == '__main__':
    main()
```

We're just about ready to go. We just need to install Matplotlib and Tkinter. Tkinter is a library that will show our Matplotlib graph in a GUI window, whether we're running Windows, Mac, or Linux. This way, we can reliably show the graph regardless of our environment:

```
pip install matplotlib tk
```

We also need to change the imports a bit. We need to tell Matplotlib to use Tkinter to display it, which can be done by specifying the library we'll use to display a window showing our Matplotlib plot. In addition, we can also ask Copilot for these imports:

JM # show me the libraries I need to import to make this script functional

You should get an output like this. If you don't know the libraries needed, you can use trial and error until you get something like

```
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use('TkAgg')
```

This way, we can specify Tkinter as our GUI library. Now it's time to run it!!

```
python main.py
```

And we did it! Here are the most used words in the novel *Free Air* presented in an easy-to-understand graph (figure 2.23).

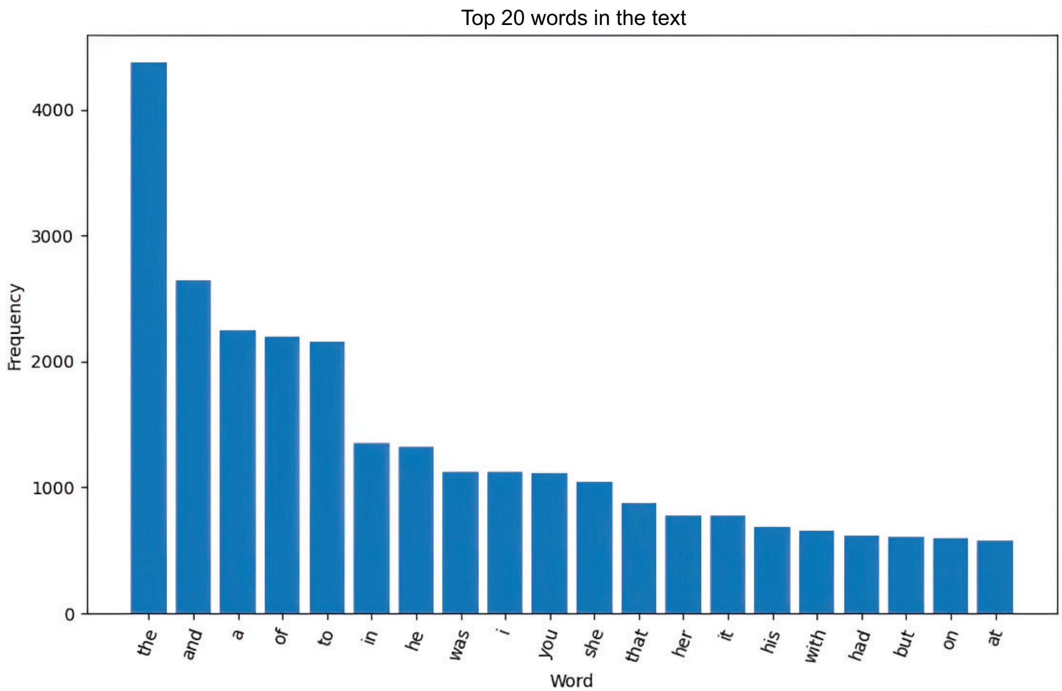

Figure 2.23 Our Matplotlib graph shows the top 20 words in the text file.

This isn't a complex application, but it shows how quickly you can use GitHub Copilot for your projects. From simple one-off scripts to large enterprise applications, I'm impressed with how well Copilot grasps context and contributes useful suggestions.

Getting started is easy with these tools. Building them into your workflow may take more time. Of course, there are many ways you can dig deeper with this tool, and we certainly will.

Summary

- Be as clear and as descriptive as possible. Copilot evaluates comments and function names for suggestions. For example, `connect_to_database` is better than `get_data`.
- Use complete sentences and clear directions. For example, "Convert the text to lowercase" is better than "make lowercase."
- Provide sufficient context when selecting code. You improve the results by selecting more than just the line of code with a problem. Surrounding text gives the request more context to work with.
- Don't create large prompts. The saying "Do one thing and do it well" applies to prompts. If they are too long and complex, they should be broken down to prevent errors.
- You can always try different prompts. If the prompt you use in the chat isn't giving you what you want, you can get creative and try different things. Make a note of the prompts that serve you well.
- Chat naturally with Copilot as if it were a human. If you get stuck, try explaining the problem like you would to another person in a real conversation. This is what the model was designed for, and it works well.
- Always review and test generated code. As demonstrated with the text cleaning functions, Copilot's first suggestion may not be the most efficient or secure solution.
- Use Copilot's different interaction methods appropriately. Code completion works well for simple tasks, while chat is better for refactoring or understanding complex code.
- Remember that context matters. Copilot analyzes your entire workspace, so keeping related files open and organized can improve its suggestions.
- Don't be afraid to iterate. As shown in the text-cleaning example, you can generate multiple solutions and test them to find the best one.
- Consider performance implications. While Copilot generates working code, you may need to profile and optimize it for your specific needs, as demonstrated in the punctuation-removal comparison.
- Include proper error-handling and edge cases. Our example showed how adding UTF-8 encoding support could make the file reading more robust across different operating systems.