IN THIS CHAPTER

- Answering the question, "Why Python for AI?"
- Reviewing the Python programming environment components
- Writing your first Python program
- Exploring useful Python functionality

Chapter 1
Starting with Python Basics

Welcome to the wonderful world of using Python with artificial intelligence (AI)! In this chapter, you get your feet wet by exploring what the Python programming language is and how it's become the leading language to work with AI tools. You get acquainted with fundamental programming concepts: text editors, the Python language interpreter, and the terminal. You write your very first Python program using Microsoft Visual Studio Code (VS Code) and Anaconda. You explore more Python concepts, including libraries, lists, loops, and functions, in your second written program. Finally, you learn some tips and tricks for debugging programs and writing great software. Ready to start exploring?

Using Python to Work with AI

The world of AI is vast, but the Python programming language has emerged as one of the most popular ways to create, test, and use AI tools. It's a good idea to understand some of the basics behind the language itself and the many ways it's used in the AI industry.

What Is Python?

Python is a programming language like C/C++, Java, and Rust. Programming languages use written commands to control underlying computer resources. Collections of written commands are called *programs* or *software*. People who write programs are called programmers, coders, software engineers, developers, computer scientists, or sleep-deprived, coffee-slurping zombies, depending on how soon a project is due.

The type of programming language you use determines how you format the commands you give the computer, what command capabilities you have at your disposal, and how the commands will actually access the computer resources you want to use. Python is a very flexible and versatile language, with a user-friendly command-writing structure (also called a *syntax*). Because the syntax is user-friendly, many people have written collections of software in Python that can be reused by other developers. These reusable pieces of code are called *libraries* or *packages*.

TECHNICAL STUFF

Libraries of reusable code are common in many programming languages, which reduces the amount of software other developers have to rewrite. Python calls its libraries *packages*, which are directories of individual Python files implementing the reusable code. Individual files within a package are called *modules*.

When your computer carries out an instruction written in a programming language, it's executing that command. Python is an *interpreted* language, which means that it uses another piece of software (the interpreter) to translate and run the written commands on your computer, usually line by line. A *compiled* language (like C/C++) first translates all the commands in the program and then runs them on your computer. Compiled languages tend to run faster since they can use the translation step to make optimizations, but interpreted languages are easy to prototype and often are simple to use on multiple types of computers.

The Python Software Foundation (`www.python.org`) maintains the language, hosts package repositories, and coordinates the development of new versions. They have excellent documentation on their website if you want to dive deeper

into the world of the language! For this book, we cover a lot of the basics of Python as well as how the language relates to AI development.

Why use Python with AI?

Out of the many, many (seriously . . . so many!) programming languages in existence, it's fair to ask, How did Python become the top contender for AI developers? It turns out, Python had a lot of the right characteristics at the right time, which include:

- Simplicity
- Numerous AI-focused packages
- High community support

Simplicity

Python's syntax — the structure of writing commands in the language — is easy to read, especially compared to many other programming languages. Many of the keywords you use when creating programs have intuitive meanings, and Python enforces a spacing scheme that makes programs easy to scan at a glance. The simple syntax of Python has made it a low-barrier-to-entry language — simply put, you don't have to be a computer scientist to learn Python. You don't even have to be a computer scientist to be good at Python! Anyone can pick it up and put it to work automating digital tasks.

Numerous AI-focused packages

Python has a huge ecosystem of libraries and packages that are useful for AI tasks. AI has a lot of interconnected components, including data preparation, model development, and model evaluation. There's a wealth of libraries written in Python that handle every component, allowing you to write less software and focus more on tailoring an AI model to your application. These include data manipulation tools like Pandas (Chapter 3), machine learning model libraries like scikit-learn (Chapter 4), and deep-learning powerhouses like Keras and PyTorch (Chapter 10). Chances are, for whatever AI-integrated task you have in mind, someone has written a Python library that takes the pain out of accomplishing it.

High community support

Having software available for AI tasks is great — unless no one bothered to explain how to use it (and if that seems ridiculous, ask any embedded systems

engineer — it happens). Python, fortunately, has a large and active user base that documents their projects, creates tutorials for packages, and helps users debug problems. All developers at some point hit a wall with a program they're writing. The Python community has made it much easier to find and diagnose problems, push through the wall, and keep going.

It's no secret that interest in AI has exploded. Python has made it possible for many different people to interact with, develop for, and push the boundaries of AI — no engineering or computer science background required. It's one of the reasons reading this book is so exciting — you get to program with these tools directly.


TECHNICAL STUFF

Python (an interpreted language) runs much more slowly than a compiled language. This has been noted as a major flaw when it comes to AI development, because many models rely on compute speed to increase their training performance. However, Python lends itself to implementing complex AI algorithms with much less overhead than other languages. A lot of optimizations have been tried and tested with Python to increase its speed in the training phase.

Setting Up a Python Programming Environment

The Appendix of this book walks you through the installation process for a Python programming environment. There are multiple pathways to working with Python, and they require different resources. However, most of them have the following components in common:

- Text editor
- Python interpreter
- Terminal
- Integrated development environment (IDE)


TIP

If this is your first time programming in a computer language, it can be a little confusing at first to figure out where each of these components lives on your computer and how each of them contributes to creating an output. We'll go over each of these individually. If you've programmed before, this is a good review, but you can also skip ahead to the next section if you have a good grasp on these concepts.

Text editor

A text editor is used to write Python commands that make up Python programs. That's it. You can use a simple document editor, a notepad app, or just about anything that lets you write text. There's no special software strictly needed to write down the Python commands you want your computer to run. However, it's common for developers to use special software to write these programs with features that make programming mistakes easier to catch and fix.

A file that contains your written Python commands is called a *Python program* or a *Python script*. Again, there's nothing special about this file, except that the name must end in the extension `.py`. This extension lets the computer know that this file is meant to be run by the Python interpreter, and it will contain commands written in the Python programming language. In most of this book, we'll use Anaconda as our Python virtual environment and interpreter, and VS Code as our IDE (which is a combination text editor and terminal). We give instructions for installing these and the relevant extensions in the appendix.

Python interpreter

The Python interpreter is the software installed on your computer that translates commands written in your Python file into actions that your computer executes. On most computers available today, this software doesn't come pre-installed so you have to find a version to put on your machine that works for your needs. You can install a Python interpreter directly on your system (especially in Linux and Mac operating systems), or you can install virtual environment software, which allows you to interact with a Python interpreter in a more managed way. The Anaconda virtual environment software is popular for managing different versions of Python interpreters and packages depending on your needs. Most of this book's examples are run in an Anaconda environment.

TIP

Virtual environments are often considered "best practice" for implementing a Python environment, especially if you are starting out. Virtual environments allow you to keep Python interpreter versions — and packages installed with them — inside a specific namespace. If you accidentally install the wrong version of a package or mess up the namespace, this lets you revert to a fresh, clean base environment. It's an extra safeguard when experimenting with new libraries.

Terminal

The terminal is a program available on most operating systems that allows you to give your computer commands. It's also the big black rectangle you see inside

every movie hacking scene ("I'm in."). The type of commands your computer accepts depends on your operating system — which is usually Windows, Mac, or some version of Linux (like Ubuntu). The terminal lets you command your computer to use the Python interpreter to execute a Python program. In most cases, you do this using the `python` keyword. To execute a Python program, you open a terminal on your computer and type:

```
python <program_name>.py
```

There are different types of terminals, but your operating system usually restricts your options. While the above command works on most operating systems to "run" your Python program, there are definitely some "gotchas" that might prevent this from happening! We'll go through these when we start programming.

You can give lots of other commands to your operating system by using the terminal. The terminal will print the output of the programs or indicate where a mistake in the program was made that prevented it from running. Being familiar with some basic terminal commands in your operating system will go a long way in beginning your programming journey. Plus, you can impress your non-programmer friends just by opening a terminal. Once, when caught off guard by a random school tour, one of Marz's students opened several programs on his computer just to look busy and "techie" for the crowd. It turns out he was just changing his monitor settings.

Integrated development environment

The last component — an integrated development environment (IDE) — is a software program that combines multiple elements together. Most commonly, an IDE will bundle a text editor for writing programs with a terminal interface for running them. The text editor will allow you to view your files and write commands in them, and will usually highlight mistakes or provide coding help to make your programming more efficient. Another part of the program has a terminal, where you can type commands into your operating system to run the Python programs. In this book, most examples use VS Code, a free IDE from Microsoft, which works for many programming languages. Other popular Python-focused IDEs include PyCharm and Spyder.

Another common Python programming IDE is Google Colab, which is available with a Google account. This allows you to use Python entirely in your browser (without installing anything extra), and instead of using a terminal, you can run code (written in code blocks or code cells) directly. We use Colab in Chapter 15

because this gives us access to advanced computer resources useful for more complex AI models. Figure 1-1 gives an overview of how the different Python programming components fit together.

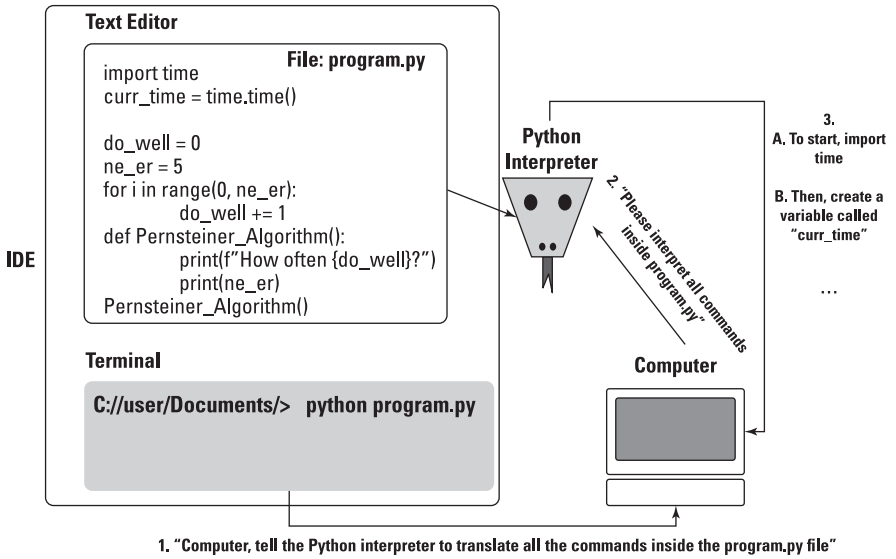

FIGURE 1-1: Python programming components and how they work together.

First-Time Python Programming Gotchas

While Python is comparatively easy to pick up, there are definitely some pitfalls to watch out for. If you run into issues, don't panic — every first-time programmer makes mistakes! The Internet is a great resource for troubleshooting, but we'll also include some resources for navigating common sources of problems.

Understanding the terminal

The terminal is a common cause of tears for new programmers. After all, you just started learning a new programming language — only to realize that in order to test your new language skills, you also have to learn how to talk to your computer in its own language! Terminals execute your commands to the operating system from a location in your computer's file system. For example, if you try to run the Python interpreter on a Python file, but your terminal is not located inside the same folder as the file, nothing will happen. Frustrating!

Figure 1-2 displays a diagram of a general terminal in most operating systems.

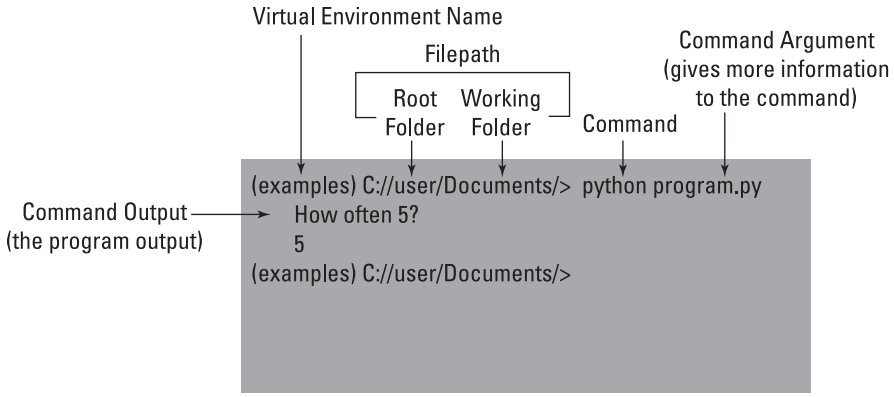

FIGURE 1-2: Parts of a terminal.

The filepath, which displays your location in the file system, is printed on the right, and you can type commands to the left of it. If you are working in a virtual environment, the name of the environment you are working in will be printed inside parentheses on the extreme right. In many IDEs (and even graphical file systems), you can open a terminal inside a given folder to start with, which helps. There are also a couple of commands that help you navigate the location inside the terminal. After you type a command in your terminal, press **Enter** on your keyboard to execute it.

To print a list of files and folders in your current terminal location, you can type:

```
ls
```

in a Linux/Mac environment, or

```
dir
```

in a Windows environment. Go ahead — open a terminal (Mac/Linux), a command prompt (Windows), or the Anaconda Prompt (Windows/Linux), and try it out!

To change the location of the terminal, you can type:

```
cd <folder_name>
```

in all three environments. If you need to go up a folder, you can type:

```
cd ..
```

So, if you have a folder structure that looks like:

```
top
        middle
                bottom
```

And your terminal is currently in the "middle" folder, you would type:

```
cd bottom
```

to change to the "bottom" folder and

```
cd ..
```

to go back to the "top" folder.

Sometimes, you have a lot of output printed in your terminal, which can be annoying. If you want to clear the terminal output, type:

```
cls
```

in a Windows environment, and

```
clear
```

in a Linux/Mac environment.

Generally, pressing the up arrow on your keyboard will bring up your most recently run command, which is a time saver (repeatedly pressing the up arrow will cycle through your command history). Pressing the tab key on your keyboard while writing out a filename will usually attempt to autocomplete the filename with matching files inside the folder.

Using different virtual environments

If you are working with a virtual environment, another common pitfall is not having the right environment active when you try to run a command. Speaking from experience, this encompasses something like 70 percent of the first-time "gotchas!" when working with Python. For an Anaconda environment, you also might

need to initialize the terminal to be able to work with Anaconda. If Anaconda is already initialized, you'll see

```
(base)
```

in front of the filepath. If that's missing, you may need to run:

```
conda init
```

inside the terminal. To change to your particular virtual environment, run:

```
conda activate <environment_name>
```

For example, if I have a virtual environment named "examples," I would run:

```
conda activate examples
```

Python packages you install are tied to a given environment. If you are getting a "package missing" error on a package you know you already installed, you are probably in the wrong environment. You can always run:

```
conda deactivate
```

to get back into your base environment.

Installing Python packages

Installing Python packages is crucial to running many of the examples in this book. The commands to do this can vary slightly depending on how you installed the Python interpreter. If you have it installed directly on your system, you can usually run:

```
pip install <package_name>
```

Or:

```
python –m pip install <package_name>
```

for a given library (Pip is a Python package manager). For Anaconda, you would run:

```
conda install <package_name>
```

For example, if you want to install the pandas Python package, you would run one of these:

```
pip install pandas
python -m pip install pandas
conda install pandas
```


TIP

Anaconda comes with a lot of packages installed by default. Usually, the package manager will tell you during the installation process if a package is already present. It typically won't reinstall the package unless you ask it to get a specific version of the software.

Creating a Virtual Environment

With the basics out of the way, you are now ready to write your first Python program. To begin, you'll create a virtual environment that you'll use to run examples in this book. You have a few options to do this. If you installed Anaconda Navigator along with Anaconda, you can do this graphically in Windows/Linux.

Open Anaconda Navigator. On the left is a side navigation bar. Click the Environments tab and then click the + sign that appears on top of the word Create at the bottom of the window (see Figure 1-3). Type **examples** in the new window that pops up, and make sure Python is checked. Then click Create to initialize the new environment (this may take a few minutes).

If you don't have Anaconda Navigator, you can open a terminal (Terminal, Command Prompt, or Anaconda Navigator, depending on your system). If the Anaconda base environment is not yet initialized (it should be in Anaconda Navigator already if you're using that program), run:

```
conda init
```

Then, to create the virtual environment, run:

```
conda create -n examples
```

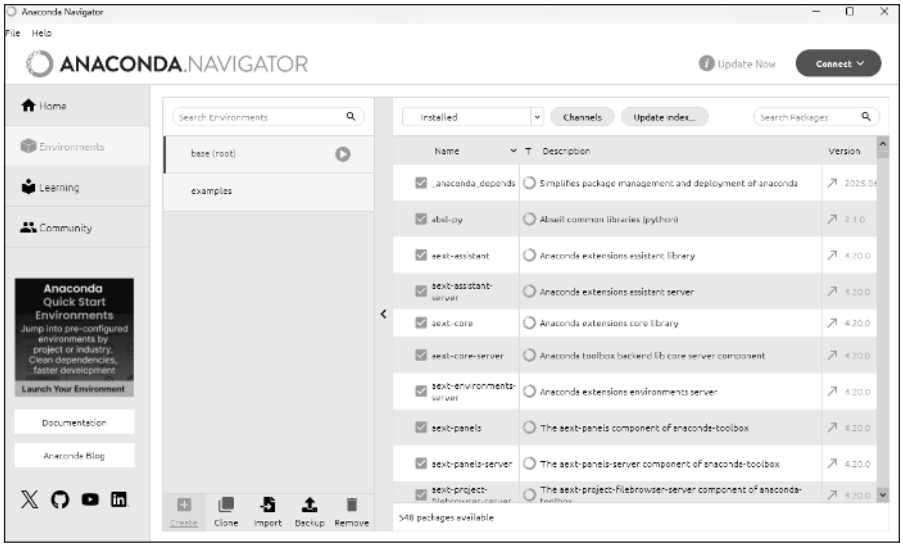

FIGURE 1-3: The Anaconda Navigator method of creating a new virtual environment.

Breaking Down the VS Code Interface

When you open VS Code for the first time after installing it, it will probably ask if you want to open a folder. Go ahead and choose a folder in your file system that you'll use to store project files for this book.

The layout for VS Code on startup is shown in Figure 1-4. To the left, you'll see the Explorer pane, which allows you to view the files located in your open folder. The icons to the extreme left allow you to switch between the Explorer pane, a Search pane, a Source Control pane, a Debugging pane, and an Extensions pane. We'll stick inside the Explorer pane for most of the book.

The top menu enables you to navigate several of the settings in VS Code. The welcome screen in the middle gives you some quick options for getting started in the IDE. Along the right is the option to chat with an AI Agent, which can help you debug, although we (ironically) won't be using it in the book.

Creating a new file

From the welcome menu, or in the Explorer pane using the + button on the file icon, create a new file. Call it **hello_world.py**. Figure 1-5 displays the view after creating the file; the file itself should be visible in the Explorer pane on the side of the screen, with the (empty) contents of the file in the middle of the screen.

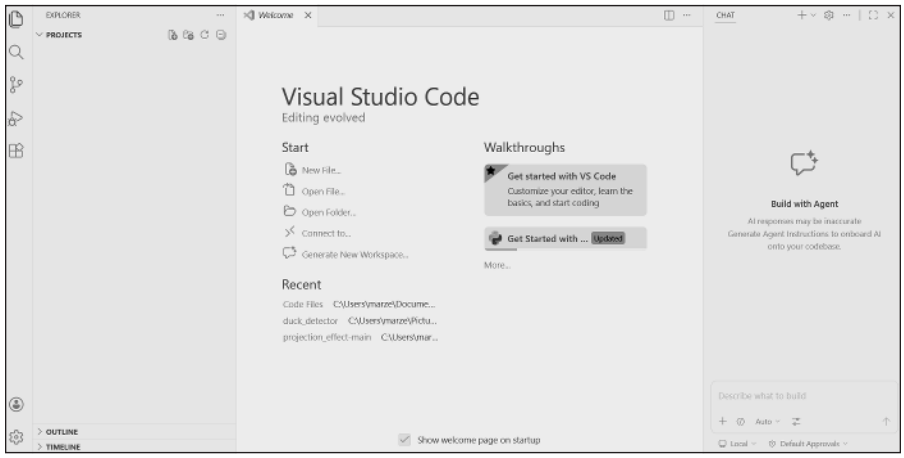

FIGURE 1-4: The VS Code start-up view.

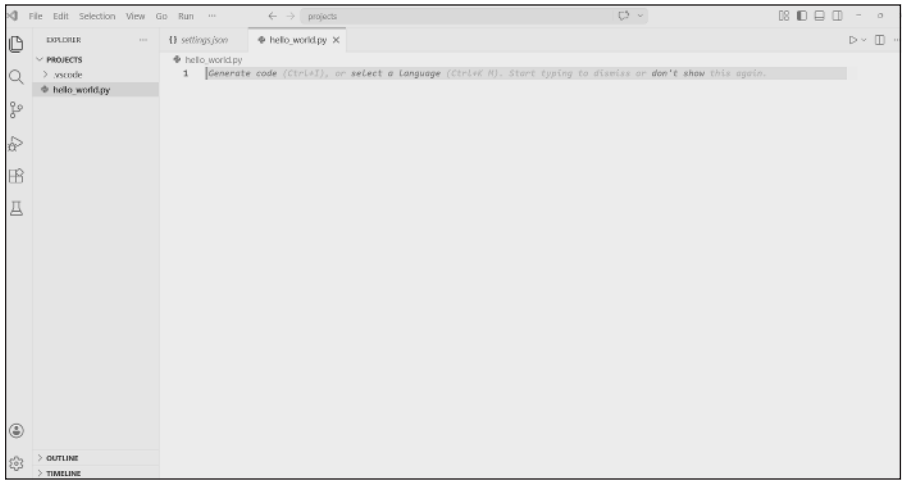

FIGURE 1-5: View in VS Code after creating a new file.

Selecting the Python interpreter

The first thing you want to do is make sure VS Code knows that you want to use Anaconda as your Python interpreter. Press Control + Shift + P in Windows/Linux or Command + Shift + P to open the Command Palette, or select View from the top menu and then Command Palette.

This opens the searchable command palette. You are going to be looking for a command that says "Python: Select Interpreter." Start typing this text, and the

matching command should show up (if it doesn't, you may need to install the Python Extension from the Extensions pane).

From the list of potential Python interpreters, select the one that matches your virtual environment name, `Python <version number> (examples)`. You should see your Python interpreter and version number at the bottom of the screen after selecting.

Opening a terminal

Now, you'll need to open a terminal. From the top menu, select Terminal (you may need to access it with the . . . menu continuation icon) and then New Terminal. This should open a new terminal at the bottom of your screen, as shown in Figure 1-6.

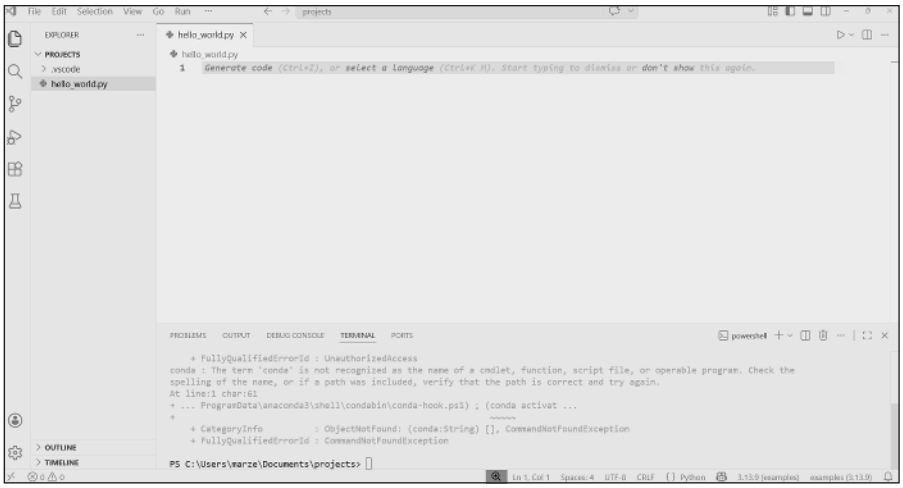

FIGURE 1-6: Opening a terminal in VS Code.

In Windows, this may open the PowerShell terminal by default, which doesn't always activate the correct Anaconda environment. If this is the case, click the down arrow by the + sign next to the PowerShell name at the top of the terminal window. Click Command Prompt. This should open the terminal and activate the correct environment (see Figure 1-7). If you see a warning about a missing temp file, you can ignore it.

Now, you're ready to start programming!

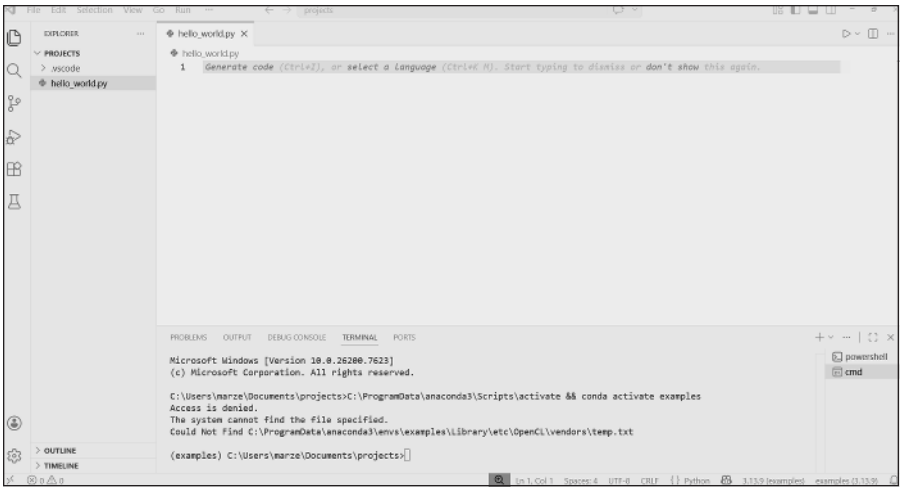

FIGURE 1-7: Using a Command Prompt terminal in VS Code.

Hello, World!

Your first Python program is incredibly simple. Listing 1-1 gives the program in its entirety.

LISTING 1-1: **Your First Python Program (hello_world.py)**

```
print("Hello, World!")
```

That's it! Remember, Python is case-sensitive, so capitalization matters — type the listing exactly as it is here. Press Control/Command + s to save the program, or click File➪ Save.

Run the program by typing:

```
python hello_world.py
```

into the terminal and press Enter. Remember, this tells your computer to use the `python` interpreter to run the commands stored in the file named `hello_world.py`.

The program only has one command — print "Hello, World." Figure 1-8 shows the terminal output you should see from running this program. Congratulations, you just commanded your computer to do something using Python!

```
(examples) C:\Users\marze\Documents\projects>python hello_world.py
Hello, World!

(examples) C:\Users\marze\Documents\projects>
```

FIGURE 1-8: Terminal output of running Listing 1-1.

TECHNICAL STUFF

Most programming languages courses start with a tutorial for printing "Hello, World!" to the terminal. It's a weird but fun computer science convention.

A more advanced program

Now, you'll create a slightly more advanced program to show off some Python functionality. Create a new file by using the + button on the file icon, or by selecting File -> New File. Name this file **more.py**. Listing 1-2 displays the commands that will be written in `more.py`, while Figure 1-9 gives the terminal output of running Listing 1-2.

LISTING 1-2: **Example Python Functionality (more.py)**

```
#Import a library
import random

#Use the library to access a function, and store the
#result in a variable
random_number = random.randint(3, 11)
#print the value of the variable
print("Random Number:")
print(random_number)

#Define a list
print("List")
list_of_numbers = [6, 5, 3, 4, 10]
#Access the first number in the list
print(list_of_numbers[0])

#Check a condition
print("Condition")
if random_number > 7:
    print("Greater than 7!")
else:
    print("Less than or equal to 7")

#Use a loop to print the numbers up to your random number
print("Loop")
```

```
for i in range(0, random_number):
    print(i)

#Define a function
def add_one(num):
    new_num = num+1
    return new_num

#Call the function
print("Call function")
returned_number = add_one(random_number)
print(returned_number)
```

```
(examples) C:\Users\marze\Documents\projects>python more.py
Random Number:
3
List
6
Condition
Less than or equal to 7
Loop
0
1
2
Call function
4
```

FIGURE 1-9: Terminal output of running Listing 1-2.

Following is a breakdown of the code, section by section:

```
#Import a library
```

This line starts with a pound sign (or hashtag, if you are from a slightly younger generation), which marks it as a comment. This means the line can be used as a note to yourself or whoever else might be reading your code — the computer won't execute it as a command.

```
import random
```

This line *imports* a package called "random" (which is installed in most Python distributions by default). It contains some handy Python code someone else has written that you can use in your project. The `import` statement allows you to

import packages that you download with `pip` or `conda`, or even functions, variables, and classes inside other Python files in the same folder!

```
#Use the library to access a function, and store the
#result in a variable
random_number = random.randint(3, 11)
#print the value of the variable
print("Random Number:")
print(random_number)
```

This section starts with comments explaining what the code is doing. We use the `randint` function located inside the `random` library (we use dot notation to access this), and *call* the function with two number *arguments* inside the parentheses. This tells the function to give us a random number between 3 and 11 (the function is inclusive, meaning it could also return 3 or 11). The result returned by the function is stored in a variable called `random_number`, which is then printed to the terminal. As shown in Figure 1-9, this function gave us the random number 3.

```
#Define a list
print("List")
list_of_numbers = [6, 5, 3, 4, 10]
#Access the first number in the list
print(list_of_numbers[0])
```

This section defines a list, which is a type of structured data in sequence. We use brackets to enclose the items in the list, and commas to separate individual items in the list. After defining the list and storing it in a variable called `list_of_numbers`, we can access different items in the list by using brackets. In this case, we access the first item in the list, index 0 (which happens to be the number 6). In Python, like many programming languages, counting items starts at 0 rather than 1.

```
#Check a condition
print("Condition")
if random_number > 7:
    print("Greater than 7!")
else:
    print("Less than or equal to 7")
```

In this section, we use Python condition-checking. The first line after the print statement checks if the `random_number` variable has a value greater than 7. If that is true, it will execute all the code in the indented block below it — in this case, a command that prints "Greater than 7!". If this is not true, it will skip down to the

`else` declaration and execute the commands indented under it, printing "Less than or equal to 7!". Since our `random_number` was given the value 3 earlier in the program, the output printed "Less than or equal to 7!". There are many possible conditions you can check in Python.

```
#Use a loop to print the numbers up to your random number
print("Loop")
for i in range(0, random_number):
    print(i)
```

This section creates a *loop*, which executes the code in the indented block multiple times. In this case, the statement creates an *iterator*, `i`, which will start at the value in the beginning of the `range` function (0) and stop one before the value at the end of the `range` function (`random_number` -1). Python is a little unusual in that the `range` function is exclusive of the second argument, meaning it will stop the loop before the iterator reaches this value. In this case, the indented block prints the value of the iterator. Since `random_number` is 3 in our case, the program prints the numbers 0, 1, and 2.

```
#Define a function
def add_one(num):
    new_num = num+1
    return new_num
```

This section defines a *function*, which is a reusable piece of code. The `def` keyword indicates that we're creating a new function. This is followed by the function name `add_one`, and open and closed parentheses which contain an argument, `num` (information given to the function when it is called). When the function is called, it will execute the code indented inside of it. In this case, it will take the `num` argument, add the number 1 to it, assign it to a variable called `new_num`, and return this value.

```
#Call the function
print("Call function")
returned_number = add_one(random_number)
print(returned_number)
```

The final section of code *calls* the function. It uses the function name (`add_one`) with the value we'll pass to it (`random_number`) inside the parentheses and returns the value of the function to a variable called `returned_number`. It then prints this `returned_number`. This takes our `random_number`, 3, and adds 1 to it to return a value of 4. We can call a function any time after it is defined.

TIP

Note that since we ask the program for a random number, we'll get a different number each time we run the program (if you are following along, you probably already noticed this).

These are some basic Python abilities to get you started, but many more resources for learning Python and testing its limits are available online. We highly encourage you to play around with it and see what's possible!

REMEMBER

Throughout this book, we provide a code listing followed by an explanation of the commands within the code. If you want to follow along on your own machine, you can find the source code for each chapter in the downloadable source code provided for this book. See the Introduction for details on how to find these source files.

USEFUL TIPS FOR WORKING WITH THE PYTHON PROGRAMMING LANGUAGE

Getting started in Python is fast, but getting comfortable with the language can take a little longer. Don't despair. This book will provide plenty of practice examples. Keep these tips in mind:

- **Be patient:** If code isn't running or a bug isn't being resolved, welcome to the programming club! A good programming practice is to never spend more than an hour or two at a time trying to solve a problem. If you're stuck, take a walk, eat a snack, or otherwise spend time giving yourself a little time off. Usually, the fix will be much faster when you're coming at it fresh.
- **Use online resources:** The Python community is a treasure trove of great libraries, excellent starter problems, and debugging help!
- **Check documentation:** If a library is behaving oddly, double-check its documentation. Sometimes a variable is supposed to be a different type than the one you're trying to use, or there are multiple similar function calls implementing different behavior.
- **Be curious:** There's no shortage of great Python projects in the wild. Read other people's code, explore different packages, and don't be afraid to implement something new!

In Chapter 2, we dive deeper into some fundamental AI concepts that you'll use in Python projects throughout the book.

Troubleshooting

It's rare to get a perfectly running program the first time around. Or the second time . . . or the third. It's okay! Persistence will pay off. Here are a few errors new programmers commonly encounter, but can be frustrating to debug:

- **File not saved:** Programs will only run saved commands. A white dot will appear on the top of the file name tab in VS Code if the file isn't saved. Make sure to save it before you try and run it!
- **Trapped inside Python terminal:** If you type the `python` command and then forget to type a file name after, you will run the interpreter directly inside the terminal. It will be expecting all subsequent commands to be in the Python language, rather than in your computer's language. Go ahead and type `exit()` and press Enter to exit this mode.
- **Syntax errors:** If you have an error with your written commands in the program, the terminal will not run your program fully. In this case, the interpreter program will generally print out the line on which the error occurred and what it thinks the error is. It can take some practice to get used to debugging these problems, but the provided information can be helpful!