2

ENVIRONMENT SETUP AND THE COMMAND LINE

Environment setup is the process of organizing your computer so you can write code. This involves installing any necessary tools, configuring them, and handling any hiccups during the setup. There is no single setup process because everyone has a different computer with a different operating system, version of the operating system, and version of the Python interpreter. Even so, this chapter describes some basic concepts to help you administer your own computer using the command line, environment variables, and filesystem.

Learning these concepts and tools might seem like a headache. You want to write code, not poke around configuration settings or understand inscrutable console commands. But these skills will save you time in the long run. Ignoring error messages or randomly changing configuration settings to get your system working well enough might hide problems, but it won't fix them. By taking the time to understand these issues now, you can prevent them from reoccurring.

The Filesystem

The *filesystem* is how your operating system organizes data to be stored and retrieved. A file has two key properties: a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer. For example, a file on my Windows 10 laptop has the filename *project.docx* in the path *C:\Users\Al\Documents*. The part of the filename after the last period is the file's *extension* and tells you a file's type. The filename *project.docx* is a Word document, and *Users*, *Al*, and *Documents* all refer to *folders* (also called *directories*). Folders can contain files and other folders. For example, *project.docx* is in the *Documents* folder, which is in the *Al* folder, which is in the *Users* folder. Figure 2-1 shows this folder organization.

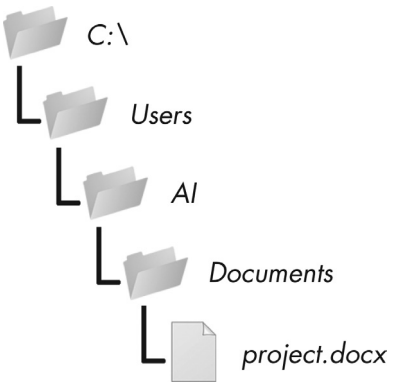

Figure 2-1: A file in a hierarchy of folders

The *C:* part of the path is the *root folder*, which contains all other folders. On Windows, the root folder is named *C:* and is also called the *C:* drive. On macOS and Linux, the root folder is */*. In this book, I'll use the Windows-style root folder, *C:*. If you're entering the interactive shell examples on macOS or Linux, enter */* instead.

Additional volumes, such as a DVD drive or USB flash drive, will appear differently on different operating systems. On Windows, they appear as new, lettered root drives, such as *D:* or *E:*. On macOS, they appear as new folders within the */Volumes* folder. On Linux, they appear as new folders within the */mnt* ("mount") folder. Note that folder names and filenames are not case sensitive on Windows and macOS, but they're case sensitive on Linux.

Paths in Python

On Windows, the backslash (\) separates folders and filenames, but on macOS and Linux, the forward slash (/) separates them. Instead of writing code both ways to make your Python scripts cross-platform compatible, you can use the `pathlib` module and `/` operator instead.

The typical way to import `pathlib` is with the statement `from pathlib import Path`. Because the `Path` class is the most frequently used class in

pathlib, this form lets you type Path instead of pathlib.Path. You can pass a string of a folder or filename to Path() to create a Path object of that folder or filename. As long as the leftmost object in an expression is a Path object, you can use the / operator to join together Path objects or strings. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

Note that because I ran this code on a Windows machine, Path() returns WindowsPath objects. On macOS and Linux, a PosixPath object is returned. (POSIX is a set of standards for Unix-like operating systems and is beyond the scope of this book.) For our purposes, there's no difference between these two types.

You can pass a Path object to any function in the Python standard library that expects a filename. For example, the function call open(Path('C:\\') / 'Users' / 'Al' / 'Desktop' / 'spam.py') is equivalent to open(r'C:\Users\Al\Desktop\spam.py').

The Home Directory

All users have a folder called the *home folder* or *home directory* for their own files on the computer. You can get a Path object of the home folder by calling Path.home():

```
>>> Path.home()
WindowsPath('C:/Users/Al')
```

The home directories are located in a set place depending on your operating system:

- On Windows, home directories are in *C:\Users*.
- On Mac, home directories are in */Users*.
- On Linux, home directories are often in */home*.

Your scripts will almost certainly have permissions to read from and write to the files in your home directory, so it's an ideal place to store the files that your Python programs will work with.

The Current Working Directory

Every program that runs on your computer has a *current working directory* (*cwd*). Any filenames or paths that don't begin with the root folder you can assume are in the cwd. Although "folder" is the more modern name for a

directory, note that cwd (or just working directory) is the standard term, not "current working folder."

You can get the cwd as a `Path` object using the `Path.cwd()` function and change it using `os.chdir()`. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> import os
❶>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python38')
❷>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')
```

Here, the cwd was set to *C:\Users\Al\AppData\Local\Programs\Python\Python38* ❶, so the filename *project.docx* would refer to *C:\Users\Al\AppData\Local\Programs\Python\Python38\project.docx*. When we change the cwd to *C:\Windows\System32* ❷, the filename *project.docx* would refer to *C:\Windows\System32\project.docx*.

Python displays an error if you try to change to a directory that doesn't exist:

```
>>> os.chdir('C:/ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:/ThisFolderDoesNotExist'
```

The `os.getcwd()` function in the `os` module is a former way of getting the cwd as a string.

Absolute vs. Relative Paths

There are two ways to specify a file path:

- An absolute path, which always begins with the root folder
- A relative path, which is relative to the program's cwd

There are also the *dot* (.) and *dot-dot* (..) folders. These are not real folders but special names that you can use in a path. A single period (.) for a folder name is shorthand for "this directory." Two periods (..) means "the parent folder."

Figure 2-2 shows an example of some folders and files. When the cwd is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.

The .\ at the start of a relative path is optional. For example, *.\spam.txt* and *spam.txt* refer to the same file.

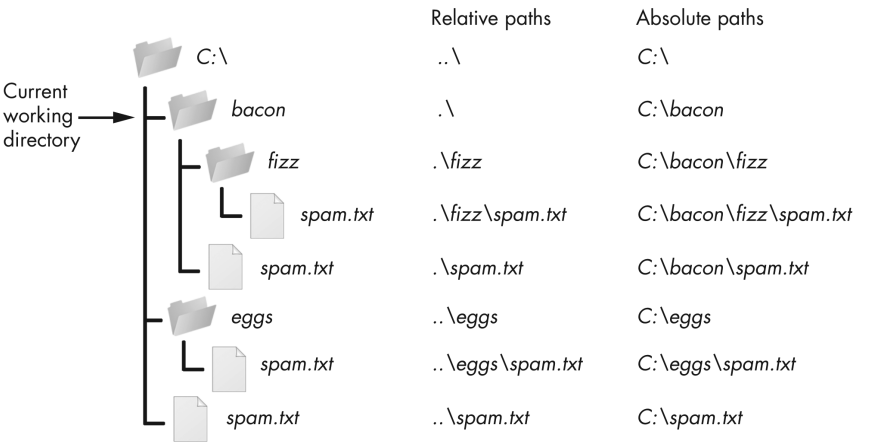

Figure 2-2: The relative paths for folders and files in the working directory C:\bacon

Programs and Processes

A *program* is any software application that you can run, such as a web browser, spreadsheet application, or word processor. A *process* is a running instance of a program. For example, Figure 2-3 shows five running processes of the same calculator program.

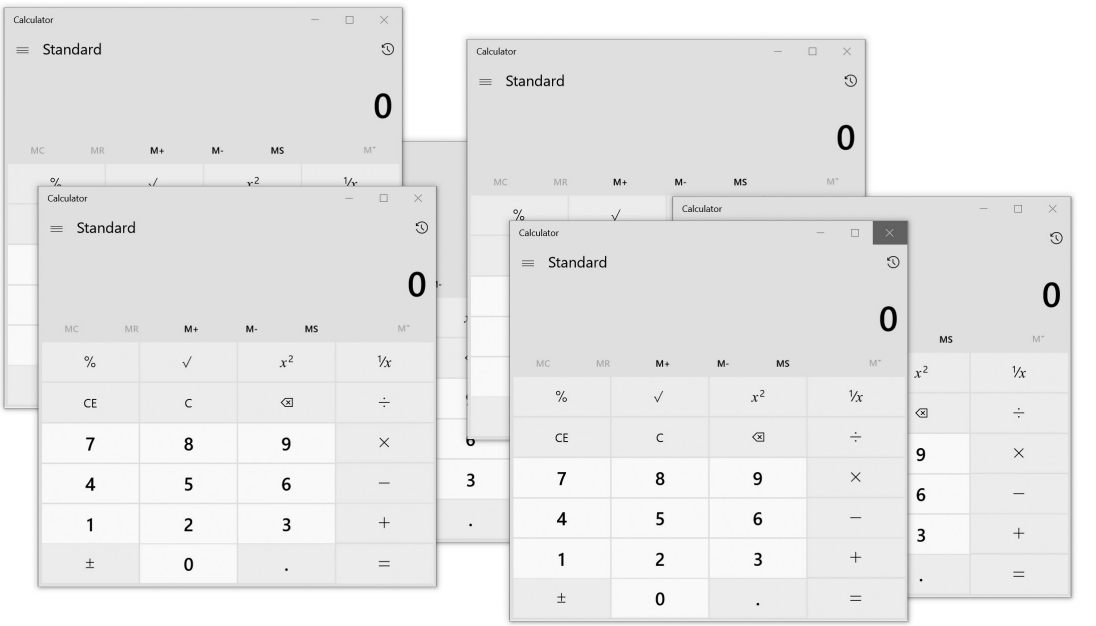

Figure 2-3: One calculator program running multiple times as multiple, separate processes

Processes remain separate from each other, even when running the same program. For example, if you ran several instances of a Python program at the same time, each process might have separate variable values. Every process, even processes running the same program, has its own cwd and environment variable settings. Generally speaking, a command line will run only one process at a time (although you can have multiple command lines open simultaneously).

Each operating system has a way of viewing a list of running processes. On Windows, you can press CTRL-SHIFT-ESC to bring up the Task Manager application. On macOS, you can run Applications ▸Utilities ▸Activity Monitor. On Ubuntu Linux, you can press CTRL-ALT-DEL to open an application also called the Task Manager. These task managers can force a running process to terminate if it's unresponsive.

The Command Line

The *command line* is a text-based program that lets you enter commands to interact with the operating system and run programs. You might also hear it called the command line interface (CLI, which rhymes with "fly"), command prompt, terminal, shell, or console. It provides an alternative to a *graphical user interface* (*GUI*, pronounced "gooey"), which allows the user to interact with the computer through more than just a text-based interface. A GUI presents visual information to a user to guide them through tasks more easily than the command line does. Most computer users treat the command line as an advanced feature and never touch it. Part of the intimidation factor is due to the complete lack of hints of how to use it; although a GUI might display a button showing you where to click, a blank terminal window doesn't remind you what to type.

But there are good reasons for becoming adept at using the command line. For one, setting up your environment often requires you to use the command line rather than the graphical windows. For another, entering commands can be much faster than clicking graphical windows with the mouse. Text-based commands are also less ambiguous than dragging an icon to some other icon. This lends them to automation better, because you can combine multiple specific commands into scripts to perform sophisticated operations.

The command line program exists in an executable file on your computer. In this context, we often call it a shell or shell program. Running the shell program makes the terminal window appear:

- On Windows, the shell program is at *C:\Windows\System32\cmd.exe*.
- On macOS, the shell program is at */bin/bash*.
- On Ubuntu Linux, the shell program is at */bin/bash*.

Over the years, programmers have created many shell programs for the Unix operating system, such as the Bourne Shell (in an executable file named *sh*) and later the Bourne-Again Shell (in an executable file named *Bash*). Linux uses Bash by default, whereas macOS uses the similar

Zsh or Z shell in Catalina and later versions. Due to its different development history, Windows uses a shell named Command Prompt. All these programs do the same thing: they present a terminal window with a text-based CLI into which the user enters commands and runs programs.

In this section, you'll learn some of the command line's general concepts and common commands. You could master a large number of cryptic commands to become a real sorcerer, but you only need to know about a dozen or so to solve most problems. The exact command names might vary slightly on different operating systems, but the underlying concepts are the same.

Opening a Terminal Window

To open a terminal window, do the following:

- On Windows, click the Start button, type **`Command Prompt`**, and then press ENTER.
- On macOS, click the **`Spotlight`** icon in the upper-right corner, type **`Terminal`**, and then press ENTER.
- On Ubuntu Linux, press the WIN key to bring up Dash, type **`Terminal`**, and press ENTER. Alternatively, use the keyboard shortcut CTRL-ALT-T.

Like the interactive shell, which displays a `>>>` prompt, the terminal displays a *shell prompt* at which you can enter commands. On Windows, the prompt will be the full path to the current folder you are in:

```
C:\Users\Al>your commands go here
```

On macOS, the prompt shows your computer's name, a colon, and the cwd with your home folder represented as a tilde (`~`). After this is your username followed by a dollar sign (`$`):

```
Als-MacBook-Pro:~ al$ your commands go here
```

On Ubuntu Linux, the prompt is similar to the macOS prompt except it begins with the username and an at (`@`) symbol:

```
al@al-VirtualBox:~$ your commands go here
```

Many books and tutorials represent the command line prompt as just `$` to simplify their examples. It's possible to customize these prompts, but doing so is beyond the scope of this book.

Running Programs from the Command Line

To run a program or command, enter its name into the command line. Let's run the default calculator program that comes with the operating system. Enter the following into the command line:

- On Windows, enter **`calc.exe`**.

- On macOS, enter **`open -a Calculator`**. (Technically, this runs the `open` program, which then runs the Calculator program.)
- On Linux, enter **`gnome-calculator`**.

Program names and commands are case sensitive on Linux but case insensitive on Windows and macOS. This means that even though you must type `gnome-calculator` on Linux, you could type `Calc.exe` on Windows and `OPEN -a Calculator` on macOS.

Entering these calculator program names into the command line is equivalent to running the Calculator program from the Start menu, Finder, or Dash. These calculator program names work as commands because the *calc.exe*, *open*, and *gnome-calculator* programs exist in folders that are included in the `PATH` environment variables. The section "Environment Variables and Path" on page 35 explains this further. But suffice it to say that when you enter a program name on the command line, the shell checks whether a program with that name exists in one of the folders listed in `PATH`. On Windows, the shell looks for the program in the cwd (which you can see in the prompt) before checking the folders in `PATH`. To tell the command line on macOS and Linux to first check the cwd, you must enter `./` before the filename.

If the program isn't in a folder listed in `PATH`, you have two options:

- Use the `cd` command to change the cwd to the folder that contains the program, and then enter the program name. For example, you could enter the following two commands:

```
cd C:\Windows\System32
calc.exe
```

- Enter the full file path for the executable program file. For example, instead of entering `calc.exe`, you could enter `C:\Windows\System32\calc.exe`.

On Windows, if a program ends with the file extension *.exe* or *.bat*, including the extension is optional: entering `calc` does the same thing as entering `calc.exe`. Executable programs in macOS and Linux often don't have file extensions marking them as executable; rather, they have the executable permission set. The section "Running Python Programs Without the Command Line" on page 39 has more information.

Using Command Line Arguments

Command line arguments are bits of text you enter after the command name. Like the arguments passed to a Python function call, they provide the command with specific options or additional directions. For example, when you run the command `cd C:\Users`, the `C:\Users` part is an argument to the `cd` command that tells `cd` to which folder to change the cwd. Or, when you run a Python script from a terminal window with the `python yourScript.py` command, the `yourScript.py` part is an argument telling the `python` program what file to look in for the instructions it should carry out.

Command line options (also called flags, switches, or simply options) are a single-letter or short-word command line arguments. On Windows, command line options often begin with a forward slash (`/`); on macOS and Linux, they begin with a single dash (`-`) or double dash (`--`). You already used the `-a` option when running the macOS command `open -a Calculator`. Command line options are often case sensitive on macOS and Linux but are case insensitive on Windows, and we separate multiple command line options with spaces.

Folders and filenames are common command line arguments. If the folder or filename has a space as part of its name, enclose the name in double quotes to avoid confusing the command line. For example, if you want to change directories to a folder called *Vacation Photos*, entering `cd Vacation Photos` would make the command line think you were passing two arguments, `Vacation` and `Photos`. Instead, you enter `cd "Vacation Photos"`:

```
C:\Users\Al>cd "Vacation Photos"

C:\Users\Al\Vacation Photos>
```

Another common argument for many commands is `--help` on macOS and Linux and `/?` on Windows. These bring up information associated with the command. For example, if you run `cd /?` on Windows, the shell tells you what the `cd` command does and lists other command line arguments for it:

```
C:\Users\Al>cd /?
Displays the name of or changes the current directory.

CHDIR [/D] [drive:][path]
CHDIR [..]
CD [/D] [drive:][path]
CD [..]

  ..   Specifies that you want to change to the parent directory.

Type CD drive: to display the current directory in the specified drive.
Type CD without parameters to display the current drive and directory.

Use the /D switch to change current drive in addition to changing current
directory for a drive.
--snip--
```

This help information tells us that the Windows `cd` command also goes by the name `chdir`. (Most people won't type `chdir` when the shorter `cd` command does the same thing.) The square brackets contain optional arguments. For example, `CD [/D] [drive:][path]` tells you that you could specify a drive or path using the `/D` option.

Unfortunately, although the `/?` and `--help` information for commands provides reminders for experienced users, the explanations can often be cryptic. They're not good resources for beginners. You're better off using

a book or web tutorial instead, such as *The Linux Command Line* by William Shotts, *Linux Basics for Hackers* by OccupyTheWeb, or *PowerShell for Sysadmins* by Adam Bertram, all from No Starch Press.

Running Python Code from the Command Line with -c

If you need to run a small amount of throwaway Python code that you run once and then discard, pass the `-c` switch to `python.exe` on Windows or `python3` on macOS and Linux. The code to run should come after the `-c` switch, enclosed in double quotes. For example, enter the following into the terminal window:

```
C:\Users\Al>python -c "print('Hello, world')"
Hello, world
```

The `-c` switch is handy when you want to see the results of a single Python instruction and don't want to waste time entering the interactive shell. For example, you could quickly display the output of the `help()` function and then return to the command line:

```
C:\Users\Al>python -c "help(len)"
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.

C:\Users\Al>
```

Running Python Programs from the Command Line

Python programs are text files that have the *.py* file extension. They're not executable files; rather, the Python interpreter reads these files and carries out the Python instructions in them. On Windows, the interpreter's executable file is *python.exe*. On macOS and Linux, it's *python3* (the original *python* file contains the Python version 2 interpreter). Running the commands `python yourScript.py` or `python3 yourScript.py` will run the Python instructions saved in a file named *yourScript.py*.

Running the py.exe Program

On Windows, Python installs a *py.exe* program in the *C:\Windows* folder. This program is identical to *python.exe* but accepts an additional command line argument that lets you run any Python version installed on your computer. You can run the `py` command from any folder, because the *C:\Windows* folder is included in the `PATH` environment variable. If you have multiple Python versions installed, running `py` automatically runs the latest version installed on your computer. You can also pass a `-3` or `-2` command line argument to run the latest Python version 3 or version 2 installed, respectively. Or you could enter a more specific version number, such as `-3.6` or `-2.7`, to run that

particular Python installation. After the version switch, you can pass all the same command line arguments to *py.exe* as you do to *python.exe*. Run the following from the Windows command line:

```
C:\Users\Al>py -3.6 -c "import sys;print(sys.version)"
3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)]

C:\Users\Al>py -2.7
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:25:58) [MSC v.1500 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The *py.exe* program is helpful when you have multiple Python versions installed on your Windows machine and need to run a specific version.

Running Commands from a Python Program

Python's `subprocess.run()` function, found in the `subprocess` module, can run shell commands within your Python program and then present the command output as a string. For example, the following code runs the `ls -al` command:

```
>>> import subprocess, locale
❶>>> procObj = subprocess.run(['ls', '-al'], stdout=subprocess.PIPE)
❷>>> outputStr = procObj.stdout.decode(locale.getdefaultlocale()[1])
>>> print(outputStr)
total 8
drwxr-xr-x  2 al al 4096 Aug  6 21:37 .
drwxr-xr-x 17 al al 4096 Aug  6 21:37 ..
-rw-r--r--  1 al al    0 Aug  5 15:59 spam.py
```

We pass the `['ls', '-al']` list to `subprocess.run()` ❶. This list contains the command name `ls`, followed by its arguments, as individual strings. Note that passing `['ls -al']` wouldn't work. We store the command's output as a string in `outputStr` ❷. Online documentation for `subprocess.run()` and `locale.getdefaultlocale()` will give you a better idea of how these functions work, but they'll work with any shell command.

Minimizing Typing with Tab Completion

Because advanced users enter commands into computers for hours a day, modern command lines offer features to minimize the amount of typing necessary. The *tab completion* feature (also called command line completion or autocomplete) lets a user type the first few characters of a folder or filename and then press the TAB key to have the shell fill in the rest of the name.

For example, when you type `cd c:\u` and press TAB on Windows, the current command checks which folders or files in *C:* begin with *u* and tab completes to `c:\Users`. It corrects the lowercase *u* to *U* as well. (On macOS and Linux, tab completion doesn't correct the casing.) If multiple folders

or filenames begin with *U* in the *C:* folder, you can continue to press TAB to cycle through all of them. To narrow down the number of matches, you could also type `cd c:\us`, which filters the possibilities to folders and filenames that begin with *us*.

Pressing the TAB key multiple times works on macOS and Linux as well. In the following example, the user typed `cd D`, followed by TAB twice:

```
al@al-VirtualBox:~$ cd D
Desktop/   Documents/ Downloads/
al@al-VirtualBox:~$ cd D
```

Pressing TAB twice after typing the `D` causes the shell to display all the possible matches. The shell gives you a new prompt with the command as you've typed it so far. At this point, you could type, say, `e` and then press TAB to have the shell complete the `cd Desktop/` command.

Tab completion is so useful that many GUI IDEs and text editors include this feature as well. Unlike command lines, these GUI programs usually display a small menu under your words as you type them, letting you select one to autocomplete the rest of the command.

Viewing the Command History

In their *command history*, modern shells also remember the commands you've entered. Pressing the up arrow key in the terminal fills the command line with the last command you entered. You can continue to press the up arrow key to find earlier commands, or press the down arrow key to return to more recent commands. If you want to cancel the command currently in the prompt and start from a fresh prompt, press CTRL-C.

On Windows, you can view the command history by running `doskey /history`. (The oddly named *doskey* program goes back to Microsoft's pre-Windows operating system, MS-DOS.) On macOS and Linux, you can view the command history by running the `history` command.

Working with Common Commands

This section contains a short list of the common commands you'll use in the command line. There are far more commands and arguments than listed here, but you can treat these as the bare minimum you'll need to navigate the command line.

Command line arguments for the commands in this section appear between square brackets. For example, `cd [destination folder]` means you should enter `cd`, followed by the name of a new folder.

Match Folder and Filenames with Wildcard Characters

Many commands accept folder and filenames as command line arguments. Often, these commands also accept names with the wildcard characters `*` and `?`, allowing you to specify multiple matching files. The `*` character

matches any number of characters, whereas the `?` character matches any single character. We call expressions that use the `*` and `?` wildcard characters *glob patterns* (short for "global patterns").

Glob patterns let you specify patterns of filenames. For example, you could run the `dir` or `ls` command to display all the files and folders in the cwd. But if you wanted to see just the Python files, `dir *.py` or `ls *.py` would display only the files that end in *.py*. The glob pattern `*.py` means "any group of characters, followed by `.py`":

```
C:\Users\Al>dir *.py
 Volume in drive C is Windows
 Volume Serial Number is DFF3-8658

 Directory of C:\Users\Al

03/24/2019  10:45 PM             8,399 conwaygameoflife.py
03/24/2019  11:00 PM             7,896 test1.py
10/29/2019  08:18 PM            21,254 wizcoin.py
               3 File(s)         37,549 bytes
               0 Dir(s)  506,300,776,448 bytes free
```

The glob pattern `records201?.txt` means "`records201`, followed by any single character, followed by `.txt`." This would match record files for the years *records2010.txt* to *records2019.txt* (as well as filenames, such as *records201X.txt*). The glob pattern `records20??.txt` would match any two characters, such as *records2021.txt* or *records20AB.txt*.

Change Directories with cd

Running `cd` *`[destination folder]`* changes the shell's cwd to the destination folder:

```
C:\Users\Al>cd Desktop

C:\Users\Al\Desktop>
```

The shell displays the cwd as part of its prompt, and any folders or files used in commands will be interpreted relative to this directory.

If the folder has spaces in its name, enclose the name in double quotes. To change the cwd to the user's home folder, enter **`cd ~`** on macOS and Linux, and **`cd %USERPROFILE%`** on Windows.

On Windows, if you also want to change the current drive, you'll first need to enter the drive name as a separate command:

```
C:\Users\Al>d:

D:\>cd BackupFiles

D:\BackupFiles>
```

To change to the parent directory of the cwd, use the `..` folder name:

```
C:\Users\Al>cd ..

C:\Users>
```

List Folder Contents with dir and ls

On Windows, the `dir` command displays the folders and files in the cwd. The `ls` command does the same thing on macOS and Linux. You can display the contents of another folder by running `dir` *`[another folder]`* or `ls` *`[another folder]`*.

The `-l` and `-a` switches are useful arguments for the `ls` command. By default, `ls` displays only the names of files and folders. To display a long listing format that includes file size, permissions, last modification timestamps, and other information, use `-l`. By convention, the macOS and Linux operating systems treat files beginning with a period as configuration files and keep them hidden from normal commands. You can use `-a` to make `ls` display all files, including hidden ones. To display both the long listing format and all files, combine the switches as `ls -al`. Here's an example in a macOS or Linux terminal window:

```
al@ubuntu:~$ ls
Desktop    Downloads         mu_code  Pictures  snap       Videos
Documents  examples.desktop  Music    Public    Templates
al@ubuntu:~$ ls -al
total 112
drwxr-xr-x 18 al   al   4096 Aug  4 18:47 .
drwxr-xr-x  3 root root 4096 Jun 17 18:11 ..
-rw-------  1 al   al   5157 Aug  2 20:43 .bash_history
-rw-r--r--  1 al   al    220 Jun 17 18:11 .bash_logout
-rw-r--r--  1 al   al   3771 Jun 17 18:11 .bashrc
drwx------ 17 al   al   4096 Jul 30 10:16 .cache
drwx------ 14 al   al   4096 Jun 19 15:04 .config
drwxr-xr-x  2 al   al   4096 Aug  4 17:33 Desktop
drwxr-xr-x  2 al   al   4096 Jun 17 18:16 Documents
```

The Windows analog to `ls -al` is the `dir` command. Here's an example in a Windows terminal window:

```
C:\Users\Al>dir
 Volume in drive C is Windows
 Volume Serial Number is DFF3-8658

 Directory of C:\Users\Al

06/12/2019  05:18 PM    <DIR>          .
06/12/2019  05:18 PM    <DIR>          ..
12/04/2018  07:16 PM    <DIR>          .android
--snip--
08/31/2018  12:47 AM            14,618 projectz.ipynb
10/29/2014  04:34 PM           121,474 foo.jpg
```

List Subfolder Contents with dir /s and find

On Windows, running `dir /s` displays the cwd's folders and their subfolders. For example, the following command displays every *.py* file in the *C:\github\ezgmail* folder and all of its subfolders:

```
C:\github\ezgmail>dir /s *.py
 Volume in drive C is Windows
 Volume Serial Number is DEE0-8982

 Directory of C:\github\ezgmail

06/17/2019  06:58 AM             1,396 setup.py
               1 File(s)          1,396 bytes

 Directory of C:\github\ezgmail\docs

12/07/2018  09:43 PM             5,504 conf.py
               1 File(s)          5,504 bytes

 Directory of C:\github\ezgmail\src\ezgmail

06/23/2019  07:45 PM            23,565 __init__.py
12/07/2018  09:43 PM                56 __main__.py
               2 File(s)         23,621 bytes

     Total Files Listed:
               4 File(s)         30,521 bytes
               0 Dir(s)  505,407,283,200 bytes free
```

The `find . -name` command does the same thing on macOS and Linux:

```
al@ubuntu:~/Desktop$ find . -name "*.py"
./someSubFolder/eggs.py
./someSubFolder/bacon.py
./spam.py
```

The `.` tells `find` to start searching in the cwd. The `-name` option tells `find` to find folders and filenames by name. The `"*.py"` tells `find` to display folders and files with names that match the `*.py` pattern. Note that the `find` command requires the argument after `-name` to be enclosed in double quotes.

Copy Files and Folders with copy and cp

To create a duplicate of a file or folder in a different directory, run **`copy`** ***`[source file or folder] [destination folder]`*** or **`cp`** ***`[source file or folder] [destination folder]`***. Here's an example in a Linux terminal window:

```
al@ubuntu:~/someFolder$ ls
hello.py  someSubFolder
al@ubuntu:~/someFolder$ cp hello.py someSubFolder
al@ubuntu:~/someFolder$ cd someSubFolder
al@ubuntu:~/someFolder/someSubFolder$ ls
hello.py
```

SHORT COMMAND NAMES

When I started learning the Linux operating system, I was surprised to find that the Windows copy command I knew well was named cp on Linux. The name "copy" was much more readable than "cp." Was a terse, cryptic name really worth saving two characters' worth of typing?

As I gained more experienced in the command line, I realized the answer is a firm "yes." We read source code more often than we write it, so using verbose names for variables and functions helps. But we type commands into the command line more often than we read them, so in this case, the opposite is true: short command names make the command line easier to use and reduce strain on your wrists.

Move Files and Folders with move and mv

On Windows, you can move a source file or folder to a destination folder by running `move` *`[source file or folder] [destination folder]`*. The `mv` *`[source file or folder] [destination folder]`* command does the same thing on macOS and Linux.

Here's an example in a Linux terminal window:

```
al@ubuntu:~/someFolder$ ls
hello.py  someSubFolder
al@ubuntu:~/someFolder$ mv hello.py someSubFolder
al@ubuntu:~/someFolder$ ls
someSubFolder
al@ubuntu:~/someFolder$ cd someSubFolder/
al@ubuntu:~/someFolder/someSubFolder$ ls
hello.py
```

The *hello.py* file has moved from *~/someFolder* to *~/someFolder/someSubFolder* and no longer appears in its original location.

Rename Files and Folders with ren and mv

Running `ren` *`[file or folder] [new name]`* renames the file or folder on Windows, and `mv` *`[file or folder] [new name]`* does so on macOS and Linux. Note that you can use the `mv` command on macOS and Linux for moving and renaming a file. If you supply the name of an existing folder for the second argument, the `mv` command moves the file or folder there. If you supply a name that doesn't match an existing file or folder, the `mv` command renames the file or folder. Here's an example in a Linux terminal window:

```
al@ubuntu:~/someFolder$ ls
hello.py  someSubFolder
```

```
al@ubuntu:~/someFolder$ mv hello.py goodbye.py
al@ubuntu:~/someFolder$ ls
goodbye.py  someSubFolder
```

The *hello.py* file now has the name *goodbye.py*.

Delete Files and Folders with del and rm

To delete a file or folder on Windows, run `del` ***`[file or folder]`***. To do so on macOS and Linux, run **`rm`** ***`[file]`*** (**`rm`** is short for, remove).

These two delete commands have some slight differences. On Windows, running `del` on a folder deletes all of its files, but not its subfolders. The `del` command also won't delete the source folder; you must do so with the `rd` or `rmdir` commands, which I'll explain in the "Delete Folders with rd and rmdir" section on page 34. Additionally, running `del` *`[folder]`* won't delete any files inside the subfolders of the source folder. You can delete the files by running `del /s /q` *`[folder]`*. The `/s` runs the `del` command on the subfolders, and the `/q` essentially means "be quiet and don't ask me for confirmation." Figure 2-4 illustrates this difference.

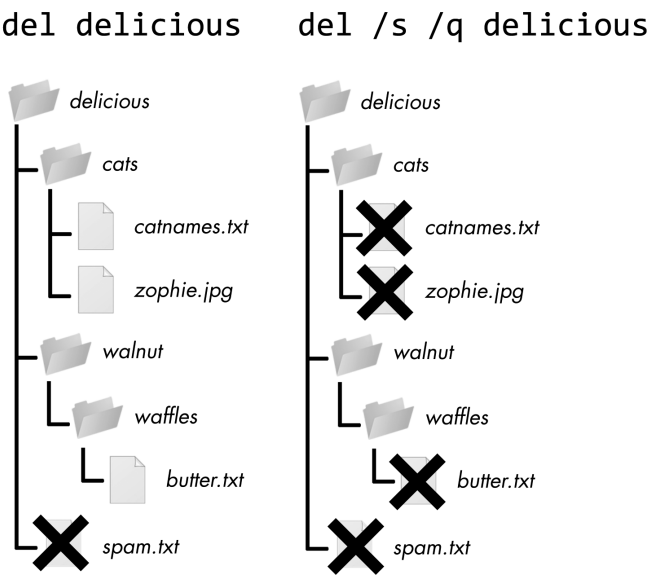

Figure 2-4: The files are deleted in these example folders when you run `del delicious` (left) or `del /s /q delicious` (right).

On macOS and Linux, you can't use the `rm` command to delete folders. But you can run `rm -r` *`[folder]`* to delete a folder and all of its contents. On Windows, `rd /s /q` *`[folder]`* will do the same thing. Figure 2-5 illustrates this task.

```
rd /s /q delicious
rm -r delicious
```

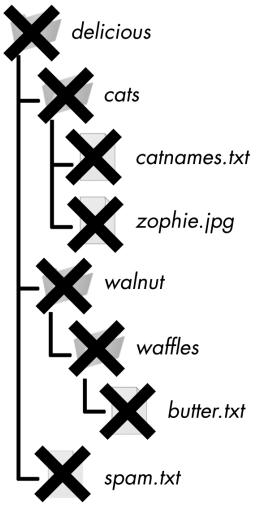

Figure 2-5: The files are deleted in these example folders when you run `rd /s /q delicious` or `rm -r delicious`.

Make Folders with md and mkdir

Running `md` *`[new folder]`* creates a new, empty folder on Windows, and running `mkdir` *`[new folder]`* does so on macOS and Linux. The `mkdir` command also works on Windows, but `md` is easier to type.

Here's an example in a Linux terminal window:

```
al@ubuntu:~/Desktop$ mkdir yourScripts
al@ubuntu:~/Desktop$ cd yourScripts
❶ al@ubuntu:~/Desktop/yourScripts$ ls
al@ubuntu:~/Desktop/yourScripts$
```

Notice that the newly created *yourScripts* folder is empty; nothing appears when we run the `ls` command to list the folder's contents ❶.

Delete Folders with rd and rmdir

Running `rd` *`[source folder]`* deletes the source folder on Windows, and `rmdir` *`[source folder]`* deletes the source folder on macOS and Linux. Like `mkdir`, the `rmdir` command also works on Windows, but `rd` is easier to type. The folder must be empty before you can remove it.

Here's an example in a Linux terminal window:

```
al@ubuntu:~/Desktop$ mkdir yourScripts
al@ubuntu:~/Desktop$ ls
yourScripts
```

```
al@ubuntu:~/Desktop$ rmdir yourScripts
al@ubuntu:~/Desktop$ ls
al@ubuntu:~/Desktop$
```

In this example, we created an empty folder named *yourScripts* and then removed it.

To delete nonempty folders (along with all the folders and files it contains), run **`rd /s/q [source folder]`** on Windows or **`rm -rf [source folder]`** on macOS and Linux.

Find Programs with where and which

Running `where` *`[program]`* on Windows or `which` *`[program]`* on macOS and Linux tells you the exact location of the program. When you enter a command on the command line, your computer checks for the program in the folders listed in the `PATH` environment variable (although Windows checks the cwd first).

These commands can tell you which executable Python program is run when you enter `python` in the shell. If you have multiple Python versions installed, your computer might have several executable programs of the same name. The one that is run depends on the order of folders in your `PATH` environment variable, and the `where` and `which` commands will output it:

```
C:\Users\Al>where python
C:\Users\Al\AppData\Local\Programs\Python\Python38\python.exe
```

In this example, the folder name indicates that the Python version run from the shell is located at *C:\Users\Al\AppData\Local\Programs\Python\Python38*.

Clear the Terminal with cls and clear

Running `cls` on Windows or `clear` on macOS and Linux will clear all the text in the terminal window. This is useful if you simply want to start with a fresh-looking terminal window.

Environment Variables and PATH

All running processes of a program, no matter the language in which it's written, have a set of variables called *environment variables* that can store a string. Environment variables often hold systemwide settings that every program would find useful. For example, the `TEMP` environment variable holds the file path where any program can store temporary files. When the operating system runs a program (such as a command line), the newly created process receives its own copy of the operating system's environment variables and values. You can change a process's environment variables independently of the operating system's set of environment variables. But those changes apply only to the process, not to the operating system or any other process.

I discuss environment variables in this chapter because one such variable, PATH, can help you run your programs from the command line.

Viewing Environment Variables

You can see a list of the terminal window's environment variables by running set (on Windows) or env (on macOS and Linux) from the command line:

```
C:\Users\Al>set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\Al\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
--snip--
USERPROFILE=C:\Users\Al
VBOX_MSI_INSTALL_PATH=C:\Program Files\Oracle\VirtualBox\
windir=C:\WINDOWS
```

The text on the left side of the equal sign (=) is the environment variable name, and the text on the right side is the string value. Every process has its own set of environment variables, so different command lines can have different values for their environment variables.

You can also view the value of a single environment variable with the echo command. Run **echo %HOMEPATH%** on Windows or **echo $HOME** on macOS and Linux to view the value of the HOMEPATH or HOME environment variables, respectively, which contain the current user's home folder. On Windows, it looks like this:

```
C:\Users\Al>echo %HOMEPATH%
\Users\Al
```

On macOS or Linux, it looks like this:

```
al@al-VirtualBox:~$ echo $HOME
/home/al
```

If that process creates another process (such as when a command line runs the Python interpreter), that child process receives its own copy of the parent process's environment variables. The child process can change the values of its environment variables without affecting the parent process's environment variables, and vice versa.

You can think of the operating system's set of environment variables as the "master copy" from which a process copies its environment variables. The operating system's environment variables change less frequently than a Python program's. In fact, most users never directly touch their environment variable settings.

Working with the PATH Environment Variable

When you enter a command, like python on Windows or python3 on macOS and Linux, the terminal checks for a program with that name in the folder

you're currently in. If it doesn't find it there, it will check the folders listed in the PATH environment variable.

For example, on my Windows computer, the *python.exe* program file is located in the *C:\Users\Al\AppData\Local\Programs\Python\Python38* folder. To run it, I have to enter `C:\Users\Al\AppData\Local\Programs\Python\Python38\python.exe`, or switch to that folder first and then enter `python.exe`.

This lengthy pathname requires a lot of typing, so instead I add this folder to the PATH environment variable. Then, when I enter `python.exe`, the command line searches for a program with this name in the folders listed in PATH, saving me from having to type the entire file path.

Because environment variables can contain only a single string value, adding multiple folder names to the PATH environment variable requires using a special format. On Windows, semicolons separate the folder names. You can view the current PATH value with the `path` command:

```
C:\Users\Al>path
C:\Path;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;
--snip--
C:\Users\Al\AppData\Local\Microsoft\WindowsApps
```

On macOS and Linux, colons separate the folder names:

```
al@ubuntu:~$ echo $PATH
/home/al/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin:/usr/games:/usr/local/games:/snap/bin
```

The order of the folder names is important. If I have two files named *someProgram.exe* in *C:\WINDOWS\system32* and *C:\WINDOWS*, entering `someProgram.exe` will run the program in *C:\WINDOWS\system32* because that folder appears first in the PATH environment variable.

If a program or command you enter doesn't exist in the cwd or any of the directories listed in PATH, the command line will give you an error, such as `command not found` or `not recognized as an internal or external command`. If you didn't make a typo, check which folder contains the program and see if it appears in the PATH environment variable.

Changing the Command Line's PATH Environment Variable

You can change the current terminal window's PATH environment variable to include additional folders. The process for adding folders to PATH varies slightly between Windows and macOS/Linux. On Windows, you can run the `path` command to add a new folder to the current PATH value:

```
❶ C:\Users\Al>path C:\newFolder;%PATH%

❷ C:\Users\Al>path
C:\newFolder;C:\Path;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;
--snip--
C:\Users\Al\AppData\Local\Microsoft\WindowsApps
```

The %PATH% part ❶ expands to the current value of the PATH environment variable, so you're adding the new folder and a semicolon to the beginning of the existing PATH value. You can run the path command again to see the new value of PATH ❷.

On macOS and Linux, you can set the PATH environment variable with syntax similar to an assignment statement in Python:

```
❶ al@al-VirtualBox:~$ PATH=/newFolder:$PATH
❷ al@al-VirtualBox:~$ echo $PATH
/newFolder:/home/al/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

The $PATH part ❶ expands to the current value of the PATH environment variable, so you're adding the new folder and a colon to the existing PATH value. You can run the echo $PATH command again to see the new value of PATH ❷.

But the previous two methods for adding folders to PATH apply only to the current terminal window and any programs run from it after the addition. If you open a new terminal window, it won't have your changes. Permanently adding folders requires changing the operating system's set of environment variables.

Permanently Adding Folders to PATH on Windows

Windows has two sets of environment variables: *system environment variables* (which apply to all users) and *user environment variables* (which override the system environment variable but apply to the current user only). To edit them, click the Start menu and then enter **`Edit environment variables for your account`**, which opens the Environment Variables window, as shown in Figure 2-6.

Select **Path** from the user variable list (not the system variable list), click **Edit**, add the new folder name in the text field that appears (don't forget the semicolon separator), and click **OK**.

This interface isn't the easiest to work with, so if you're frequently editing environment variables on Windows, I recommend installing the free Rapid Environment Editor software from *https://www.rapidee.com/*. Note that after installing it, you must run this software as the administrator to edit system environment variables. Click the Start menu, type **`Rapid Environment Editor`**, right-click the software's icon, and click **Run as administrator**.

From the Command Prompt, you can permanently modify the system PATH variable using the setx command:

```
C:\Users\Al>setx /M PATH "C:\newFolder;%PATH%"
```

You'll need to run the Command Prompt as the administrator to run the setx command.

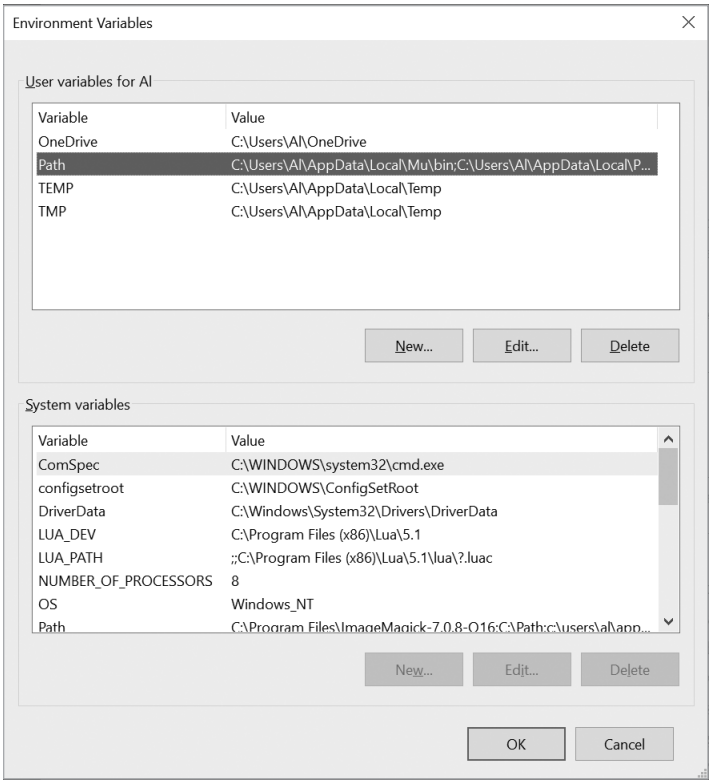

Figure 2-6: The Environment Variables window on Windows

Permanently Adding Folders to PATH on macOS and Linux

To add folders to the `PATH` environment variables for all terminal windows on macOS and Linux, you'll need to modify the *.bashrc* text file in your home folder and add the following line:

```
export PATH=/newFolder:$PATH
```

This line modifies `PATH` for all future terminal windows. On macOS Catalina and later versions, the default shell program has changed from Bash to Z Shell, so you'll need to modify *.zshrc* in the home folder instead.

Running Python Programs Without the Command Line

You probably already know how to run programs from whatever launcher your operating system provides. Windows has the Start menu, macOS has the Finder and Dock, and Ubuntu Linux has Dash. Programs will add themselves to these launchers when you install them. You can also double-click a program's icon in a file explorer app (such as File Explorer on Windows, Finder on macOS, and Files on Ubuntu Linux) to run them.

But these methods don't apply to your Python programs. Often, double-clicking a *.py* file will open the Python program in an editor or IDE instead of running it. And if you try running Python directly, you'll just open the Python interactive shell. The most common way of running a Python program is opening it in an IDE and clicking the Run menu option or executing it in the command line. Both methods are tedious if you simply want to launch a Python program.

Instead, you can set up your Python programs to easily run them from your operating system's launcher, just like other applications you've installed. The following sections detail how to do this for your particular operating system.

Running Python Programs on Windows

On Windows, you can run Python programs in a few other ways. Instead of opening a terminal window, you can press WIN-R to open the Run dialog and enter `py C:\path\to\yourScript.py`, as shown in Figure 2-7. The *py.exe* program is installed at *C:\Windows\py.exe*, which is already in the `PATH` environment variable, and the *.exe* file extension is optional when you are running programs.

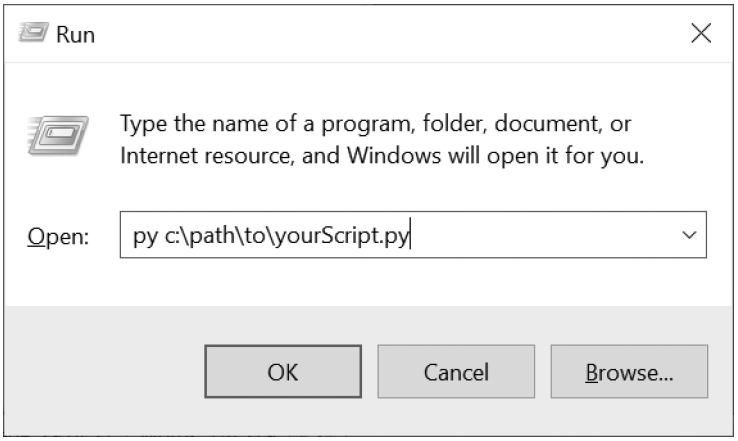

Figure 2-7: The Run dialog on Windows

Still, this method requires you to enter your script's full path. Also, the terminal window that displays the program's output will automatically close when the program ends, and you might miss some output.

You can solve these problems by creating a *batch script*, which is a small text file with the *.bat* file extension that can run multiple terminal commands at once, much like a shell script in macOS and Linux. You can use a text editor, such as Notepad, to create these files. Make a new text file containing the following two lines:

```
@py.exe C:\path\to\yourScript.py %*
@pause
```

Replace this path with the absolute path to your program, and save this file with a *.bat* file extension (for example, *yourScript.bat*). The `@` sign at the start of each command prevents it from being displayed in the terminal window, and the `%*` forwards any command line arguments entered after the batch filename to the Python script. The Python script, in turn, reads the command line arguments in the `sys.argv` list. This batch file will spare you from having to type the Python program's full absolute path every time you want to run it. The `@pause` command adds `Press any key to continue...` to the end of the Python script to prevent the program's window from disappearing too quickly.

I recommend you place all of your batch and *.py* files in a single folder that already exists in the `PATH` environment variable, such as your home folder at *C:\Users\<USERNAME>*. With a batch file set up, you can run your Python script by simply pressing WIN-R, entering the name of your batch file (entering the *.bat* file extension is optional), and pressing ENTER.

Running Python Programs on macOS

On macOS, you can create a shell script to run your Python scripts by creating a text file with the *.command* file extension. Make one in a text editor, such as TextEdit, and add the following content:

```
#!/usr/bin/env bash
python3 /path/to/yourScript.py
```

Save this file in your home folder. In a terminal window, make this shell script executable by running `chmod u+x yourScript.command`. Now you should be able to click the Spotlight icon (or press COMMAND-SPACE) and enter the name of your shell script to run it. The shell script, in turn, will run your Python script.

Running Python Programs on Ubuntu Linux

There isn't a quick way to run your Python scripts on Ubuntu Linux like there is in Windows and macOS, although you can shorten some of the steps involved. First, make sure your *.py* file is in your home folder. Second, add this line as the first line of your *.py* file:

```
#!/usr/bin/env python3
```

This is called a *shebang line*, and it tells Ubuntu that when you run this file, you want to use `python3` to run it. Third, add the execute permission to this file by running the `chmod` command from the terminal:

```
al@al-VirtualBox:~$ chmod u+x yourScript.py
```

Now whenever you want to quickly run your Python script, you can press CTRL-ALT-T to open a new terminal window. This terminal will be

set to the home folder, so you can simply enter `./yourScript.py` to run this script. The `./` is required because it tells Ubuntu that *yourScript.py* exists in the cwd (the home folder, in this case).

Summary

Environment setup involves all the steps necessary to get your computer into a state where you can easily run your programs. It requires you to know several low-level concepts about how your computer works, such as the filesystem, file paths, processes, the command line, and environment variables.

The filesystem is how your computer organizes all the files on your computer. A file is a complete, absolute file path or a file path relative to the cwd. You'll navigate the filesystem through the command line. The command line has several other names, such as terminal, shell, and console, but they all refer to the same thing: the text-based program that lets you enter commands. Although the command line and the names of common commands are slightly different between Windows and macOS/Linux, they effectively perform the same tasks.

When you enter a command or program name, the command line checks the folders listed in the `PATH` environment variable for the name. This is important to understand to figure out any `command not found` errors you might encounter. The steps for adding new folders to the `PATH` environment variable are also slightly different between Windows and macOS/Linux.

Becoming comfortable with the command line takes time because there are so many commands and command line arguments to learn. Don't worry if you spend a lot of time searching for help online; this is what experienced software developers do every day.