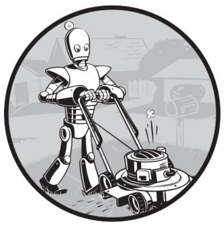
10

READING AND WRITING FILES



Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file. You can think of a file's contents as a single string value, potentially gigabytes in size. In this chapter, you'll learn how to use Python to create, read, and save files on the hard drive.

Files and Filepaths

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, there is a file on my Windows laptop with the filename *project.docx* in the path *C:\Users\Al\Documents*. The part of the filename after the last period is called 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 (called *subfolders*). For example, *project.docx* is in the *Documents* folder, which is inside the *Al* folder, which is inside the *Users* folder. Figure 10-1 shows this folder organization.

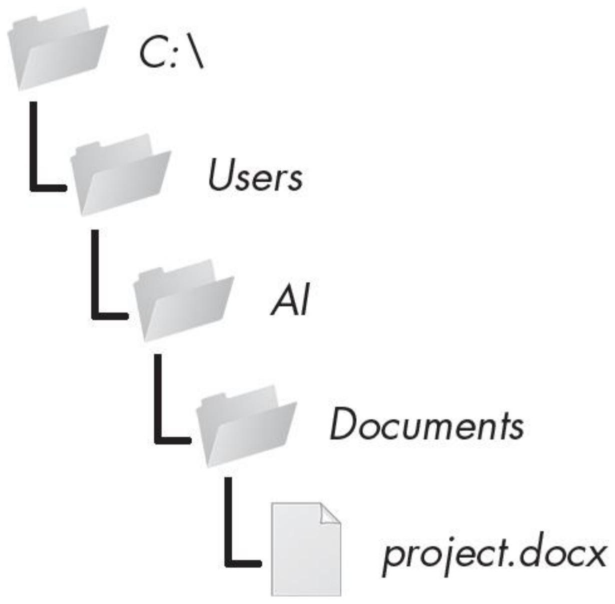


Figure 10-1: A file in a hierarchy of folders

The *C:\* part of the path is the *root folder*, which contains all the 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 are 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 under the */Volumes* folder. On Linux, they appear as new folders under the */mnt* ("mount") folder. Also note that while folder names and filenames are not case-sensitive on Windows and macOS, they are case-sensitive on Linux.

NOTE

Because your system probably has different files and folders on it than mine, you won't be able to follow every example in this chapter exactly. Still, try to follow along using folders that exist on your computer.

Standardizing Path Separators

On Windows, paths are written using backslashes (`\`) as the separator between folder names. The macOS and Linux operating systems, however, use the forward slash (/) as their path separator.

The `Path()` function in the `pathlib` module handles all operating systems, so the best practice is to use forward slashes in your Python code. If you pass it the string values of individual file and folder names in your path, `Path()` will return a string with a filepath using the correct path separators. Enter the following into the interactive shell:

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

While the `WindowsPath` object may use `/` forward slashes, converting it to a string with the `str()` function requires using `\` backslashes. Note that the convention for importing `pathlib` is to run `from pathlib import Path`, since otherwise we'd have to enter `pathlib.Path` everywhere `Path` shows up in our code. Not only is this extra typing redundant, but it's also redundant.

I'm running this chapter's interactive shell examples on Windows, so `Path('spam', 'bacon', 'eggs')` returned a `WindowsPath` object for the joined path, represented as `WindowsPath('spam/bacon/eggs')`. Even though Windows uses backslashes, the `WindowsPath` representation in the interactive shell displays them using forward slashes, as open source software developers have historically favored the Linux operating system.

If you want to get a simple text string of this path, you can pass it to the `str()` function, which in our example returns `'spam\\bacon\\eggs'`. (Notice that we double the backslashes because we need to escape each backslash with another backslash character.) If I had called this function on macOS or Linux, `Path()` would have returned a `PosixPath` object that, when passed to `str()`, would have returned `'spam/bacon/eggs'`. (*POSIX* is a set of standards for Unix-like operating systems.)

If you work with `Path` objects, `WindowsPath` and `PosixPath` never have to appear in your source code directly. These `Path` objects will be passed to several of the file-related functions introduced in this chapter. For example, the following code joins names from a list of filenames to the end of a folder's name:

```
>>> from pathlib import Path
>>> my_files = ['accounts.txt',
'details.csv', 'invite.docx']
>>> for filename in my_files:
...     print(Path(r'C:\Users\Al', filename))
...
```

```
C:\Users\Al\accounts.txt
C:\Users\Al\details.csv
C:\Users\Al\invite.docx
```

On Windows, the backslash separates directories, so you can't use it in filenames. However, you can use backslashes in filenames on macOS and Linux. So, while `Path(r'spam\eggs')` refers to two separate folders (or a file *eggs* in a folder *spam*) on Windows, the same command would refer to a single folder (or file) named *spam\eggs* on macOS and Linux. For this reason, it's usually a good idea to always use forward slashes in your Python code (and I'll be doing so for the rest of this chapter). The `pathlib` module will ensure that your code always works on all operating systems.

Joining Paths

We normally use the + operator to add two integer or floating-point numbers, such as in the expression `2 + 2`, which evaluates to the integer value `4`. But we can also use the + operator to concatenate two string values, like the expression `'Hello' + 'World'`, which evaluates to the string value `'HelloWorld'`. Similarly, the `/` operator that we normally use for division can combine `Path` objects and strings. This is helpful for modifying a `Path` object after you've already created it with the `Path()` function.

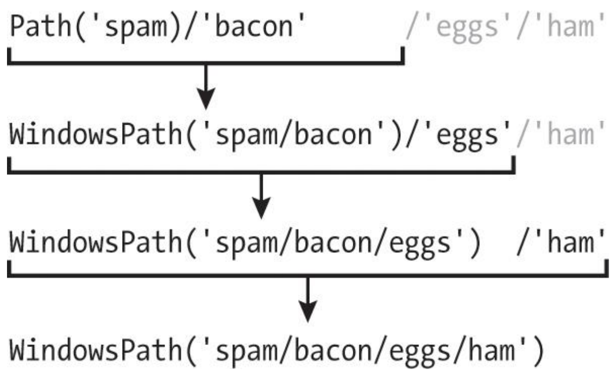
For example, enter the following into the interactive shell:

```
>>> 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')
```

The only thing you need to keep in mind when using the `/` operator for joining paths is that one of the first two values in the expression must be a `Path` object. This is because these expressions evaluate from left to right, and the `/` operator can be used on two `Path` objects or on a `Path` object and a string, but not on two strings. Python will give you an error if you try to enter the following into the interactive shell:

```
>>> 'spam' / 'bacon'
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
TypeError: unsupported operand type(s) for /:
'str' and 'str'
```

So, either the first or second leftmost value must be a `Path` object for the entire expression to evaluate to a `Path` object. Here's how the `/` operator and a `Path` object evaluate to the final `Path` object:


Path('spam)/'bacon'
/'eggs'/'ham'
WindowsPath('spam/bacon')/'eggs'/'ham'
WindowsPath('spam/bacon/eggs') /'ham'
WindowsPath('spam/bacon/eggs/ham')

Description

If you see the `TypeError: unsupported operand type(s) for /: 'str' and 'str'` error message shown previously, you need to put a `Path` object instead of a string on the left side of the expression.

The `/` operator replaces the older `os.path.join()` function, which you can learn more about at *https://docs.python.org/3/library/os.path.html#os.path.join*.

Accessing the Current Working Directory

Every program that runs on your computer has a *current working directory*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.

NOTE

While folder *is the more modern name for* directory, *note that* current working directory *(or just* working directory*) is the standard term, not* current working folder.

You can get the current working directory as a string value with the `Path.cwd()` function and can change it using `os.chdir()`. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> import os
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/
Programs/Python/Python313')'
>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')
```

Here, the current working directory is set to *C:\Users\Al\AppData\Local\Programs\Python\Python313*, so the filename *project.docx* refers to *C:\Users\Al\AppData\Local\Programs\Python\Python313\project.docx*. When we change the current working directory to *C:\Windows\System32*, the filename *project.docx* is interpreted as *C:\Windows\System32\project.docx*.

Python will display an error if you try to change to a directory that does not exist:

```
>>> import os
>>> os.chdir('C:/ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
FileNotFoundError: [WinError 2] The system
cannot find the file specified:
'C:/ThisFolderDoesNotExist'
```

There is no `pathlib` function for changing the working directory. You must use `os.chdir()`.

The `os.getcwd()` function is the older way of getting the current working directory as a string. It's documented at *https://docs.python.org/3/library/os.html#os.getcwd*.

Accessing the Home Directory

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

```
>>> from pathlib import Path
>>> 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 under *C:\Users*.
- On macOS, home directories are under */Users*.
- On Linux, home directories are often under */home*.

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

Specifying Absolute vs. Relative Paths

There are two ways to specify a filepath:

- An *absolute path*, which always begins with the root folder (*C:\* on Windows and */* on macOS and Linux)
- A *relative path*, which is relative to the program's current working directory

On Windows, *C:\* is the root for the main hard drive. This lettering dates back to the 1960s, when computers had two floppy disk drives labeled *A:\* and *B:\*. On Windows, USB flash memory and DVD drives are assigned to letters *D:\* and higher. Use one of these drives as the root folder to access files on that storage media.

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

Figure 10-2 shows some example folders and files. When the current working directory is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.

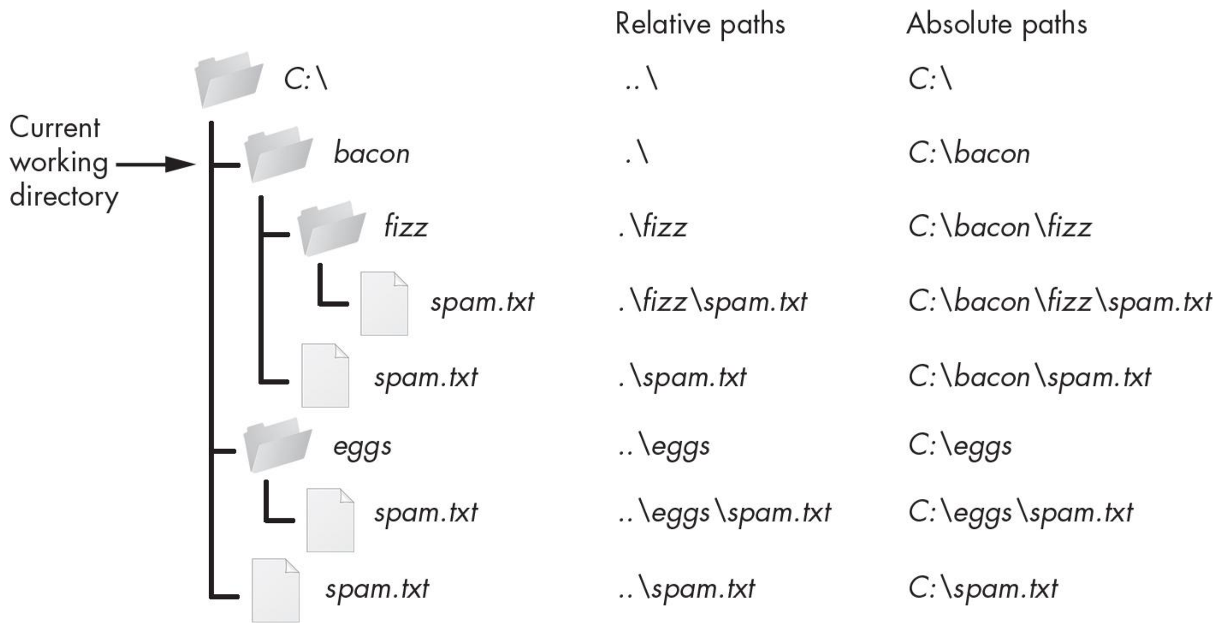


Figure 10-2: The relative paths for folders and files in the working directory C:\bacon Description

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

Creating New Folders

Your programs can create new folders with the `os.makedirs()` function. Enter the following into the interactive shell:

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\
\waffles')
```

This will create not just the *C:\delicious* folder but also a *walnut* folder inside *C:\delicious* and a *waffles* folder inside *C:\delicious\walnut*. That is, `os.makedirs()` will create any necessary intermediate folders to ensure that the full path exists. Figure 10-3 shows this hierarchy of folders.

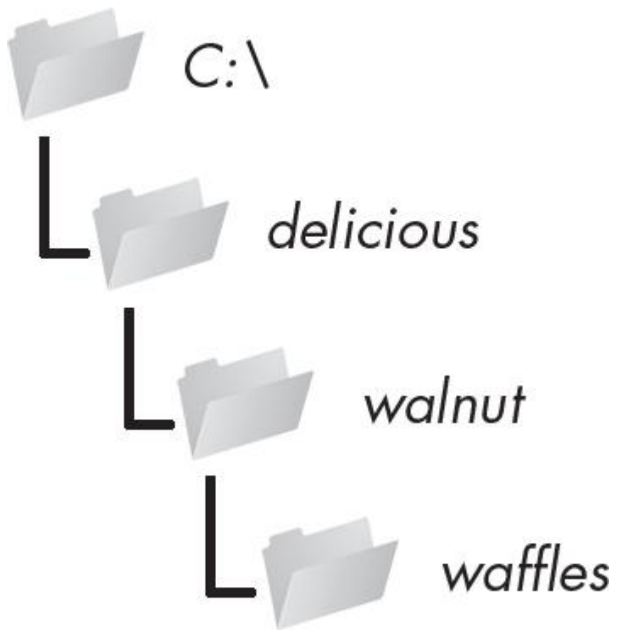


Figure 10-3: The result of `os.makedirs('C:\\delicious\\walnut\\waffles')`

To make a directory from a `Path` object, call the `mkdir()` method. For example, this code will create a *spam* folder under the home folder on my computer:

```
>>> from pathlib import Path
>>> Path(r'C:\Users\Al\spam').mkdir()
```

Note that `mkdir()` can only make one directory at a time unless you pass `parents=True`, in which case it creates all the necessary parent folders as well.

Handling Absolute and Relative Paths

Calling the `is_absolute()` method on a `Path` object will return `True` if it represents an absolute path or `False` if it represents a relative path. For example, enter the following into the interactive shell, using your own files and folders instead of the exact ones listed here:

```
>>> from pathlib import Path
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/
Programs/Python/Python312')
>>> Path.cwd().is_absolute()
True
```

```
>>> Path('spam/bacon/eggs').is_absolute()
False
```

To get an absolute path from a relative path, you can put `Path.cwd() /` in front of the relative `Path` object. After all, when we say "relative path," we almost always mean a path that is relative to the current working directory. The `absolute()` method also returns this `Path` object. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.cwd() / Path('my/relative/path')
WindowsPath('C:/Users/Al/Desktop/my/relative/
path')
>>> Path('my/relative/path').absolute()
WindowsPath('C:/Users/Al/Desktop/my/relative/
path')
```

If your relative path is relative to another path besides the current working directory, replace `Path.cwd()` with that other path. The following example gets an absolute path using the home directory instead of the current working directory:

```
>>> from pathlib import Path
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.home() / Path('my/relative/path')
WindowsPath('C:/Users/Al/my/relative/path')
```

`Path` objects are used to represent both relative and absolute paths. The only difference is whether the `Path` object begins with the root folder or not.

Getting the Parts of a Filepath

Given a `Path` object, you can extract the filepath's different parts as strings using several `Path` object attributes. These can be useful for constructing new filepaths based on existing ones. Figure 10-4 diagrams the attributes.

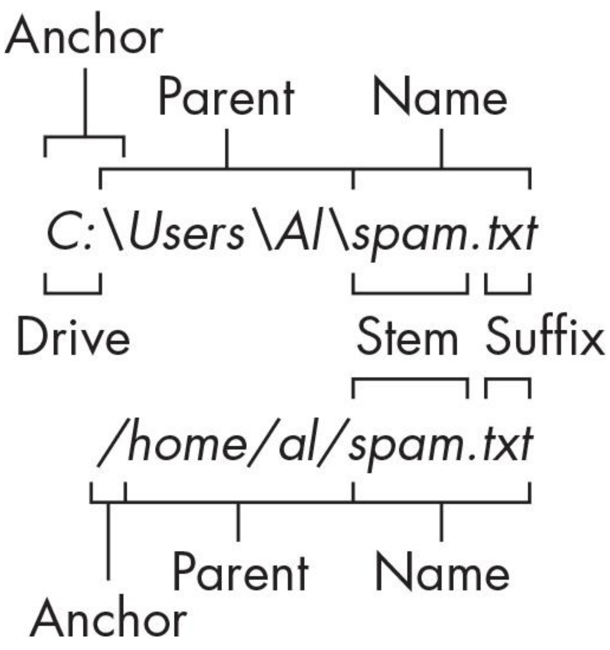


Figure 10-4: The parts of a Windows (top) and macOS/Linux (bottom) filepath Description

The parts of a filepath include the following:

- The *anchor*, which is the root folder of the filesystem
- On Windows, the *drive*, which is the single letter that often denotes a physical hard drive or other storage device
- The *parent*, which is the folder that contains the file
- The *name* of the file, made up of the *stem* (or *base name*) and the *suffix* (or *extension*)

Note that Windows `Path` objects have a `drive` attribute, but macOS and Linux `Path` objects don't. The `drive` attribute doesn't include the first backslash.

To extract each attribute from the filepath, enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.anchor
'C:\\'
>>> p.parent
```

```
WindowsPath('C:/Users/Al')
>>> p.name
'spam.txt'
>>> p.stem
'spam'
>>> p.suffix
'.txt'
>>> p.drive
'C:'
```

These attributes evaluate to simple string values, except for `parent`, which evaluates to another `Path` object. If you want to split up a path by its separator, access the `parts` attribute to get a tuple of string values:

```
>>> from pathlib import Path
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.parts
('C:\\', 'Users', 'Al', 'spam.txt')
 >>> p.parts[3]
'spam.txt'
>>> p.parts[0:2]
('C:\\', 'Users')
```

Note that even though the string used in the `Path()` call contains forward slashes, `parts` uses an anchor on Windows that has the appropriate backslash: `'C:\\'` (or `r'C:\'` as a raw string with the backslash unescaped).

The `parents` attribute (which is different from the `parent` attribute) evaluates to the ancestor folders of a `Path` object with an integer index:

```
>>> from pathlib import Path
>>> Path.cwd()
WindowsPath('C:/Users/Al/Desktop')
```

```
>>> Path.cwd().parents[0]
WindowsPath('C:/Users/Al')
>>> Path.cwd().parents[1]
WindowsPath('C:/Users')
>>> Path.cwd().parents[2]
WindowsPath('C:/')
```

If you keep following the parent folders, you will end up with the root folder.

Finding File Sizes and Timestamps

Once you have ways to handle filepaths, you can start gathering information about specific files and folders. The `stat()` method returns a `stat_result` object with file size and timestamp information about a file.

For example, enter the following into the interactive shell to find out about the *calc.exe* program file on Windows:

```
>>> from pathlib import Path
>>> calc_file = Path('C:/Windows/System32/
calc.exe')
>>> calc_file.stat()
os.stat_result(st_mode=33279,
st_ino=562949956525418, st_dev=3739257218,
st_nlink=2, st_uid=0, st_gid=0,
st_size=27648, st_atime=1678984560,
st_mtime=1575709787, st_ctime=1575709787)
>>> calc_file.stat().st_size
27648
>>> calc_file.stat().st_mtime
1712627129.0906117
>>> import time
>>> time.asctime(time.localtime(calc_file.sta
t().st_mtime))
'Mon Apr  8 20:45:29 2024'
```

The `st_size` attribute of the `stat_result` object returned by the `stat()` method is the size of the file in bytes. You can divide this integer by `1024`, by `1024 ** 2`, or by `1024 ** 3` to get the size in KB, MB, or GB, respectively.

The `st_mtime` is the "last modified" timestamp, which can be useful for, say, figuring out the last time a *.docx* Word file was changed. This timestamp is in Unix epoch time, which is the number of seconds since January 1, 1970. The `time` module (explained in Chapter 19) has functions for turning this number into a human-readable form.

The `stat_result` object has several useful attributes:

`st_size` The size of the file in bytes.

`st_mtime` The "last modified" timestamp, when the file was last changed.

`st_ctime` The "creation" timestamp. On Windows, this identifies when the file was created. On macOS and Linux, this identifies the last time the file's metadata (such as its name) was changed.

`st_atime` The "last accessed" timestamp, when the file was last read.

Keep in mind that the modified, creation, and access timestamps can be changed manually, and are not guaranteed to be accurate.

Finding Files Using Glob Patterns

The `*` and `?` characters can be used to match folder names and filenames in what are called *glob patterns*. Glob patterns are like a simplified regex language: the `*` character matches any text, and the `?` character matches exactly one character. For example, look at these glob patterns:

`'*.txt'` matches all files that end with *.txt*.

`'project?.txt'` matches `'project1.txt'`, `'project2.txt'`, or `'projectX.txt'`.

`'*project?.*'` matches `'catproject5.txt'` or `'secret_project7.docx'`.

`'*'` matches all filenames.

`Path` objects of folders have a `glob()` method for listing any content in the folder that matches the glob pattern. The `glob()` method returns a generator object (the topic of which is beyond the scope of this book) that you'll need to pass to `list()` to easily view in the interactive shell:

```
>>> from pathlib import Path
>>> p = Path('C:/Users/Al/Desktop')
>>> p.glob('*')
<generator object Path.glob at
0x000002A6E389DED0>
>>> list(p.glob('*'))
[WindowsPath('C:/Users/Al/Desktop/1.png'),
WindowsPath('C:/Users/Al/
Desktop/22-ap.pdf'), WindowsPath('C:/Users/
Al/Desktop/cat.jpg'),
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

You can also use the generator object that `glob()` returns in a `for` loop:

```
>>> from pathlib import Path
>>> for name in Path('C:/Users/Al/
Desktop').glob('*'):
>>>     print(name)
C:\Users\Al\Desktop\1.png
C:\Users\Al\Desktop\22-ap.pdf
C:\Users\Al\Desktop\cat.jpg
C:\Users\Al\Desktop\zzz.txt
```

If you want to perform an operation on every file in a folder, such as copying it to a backup folder or renaming it, the `glob('*')` method call can get you the list of `Path` objects for these files and folders. Note that glob patterns are also commonly used in command line commands such as `ls` or `dir`. Chapter 12 discusses the command line in more detail.

Checking Path Validity

Many Python functions will crash with an error if you supply them with a path that does not exist. Luckily, `Path` objects have methods to check whether a given path exists and whether it is a file or folder. Assuming that a variable `p` holds a `Path` object, you could expect the following:

- Calling `p.exists()` returns `True` if the path exists, and returns `False` if it doesn't exist.
- Calling `p.is_file()` returns `True` if the path exists and is a file, and returns `False` otherwise.
- Calling `p.is_dir()` returns `True` if the path exists and is a directory, and returns `False` otherwise.

On my computer, here is what I get when I try these methods in the interactive shell:

```
>>> from pathlib import Path
>>> win_dir = Path('C:/Windows')
>>> not_exists_dir = Path('C:/This/Folder/
Does/Not/Exist')
>>> calc_file_path = Path('C:/Windows
/System32/calc.exe')
>>> win_dir.exists()
True
>>> win_dir.is_dir()
True
>>> not_exists_dir.exists()
False
>>> calc_file_path.is_file()
True
>>> calc_file_path.is_dir()
False
```

You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the `exists()` method. For instance, if I wanted to check for a flash drive with the volume named *D:\* on my Windows computer, I could do that with the following:

```
>>> from pathlib import Path
>>> d_drive = Path('D:/')
>>> d_drive.exists()
False
```

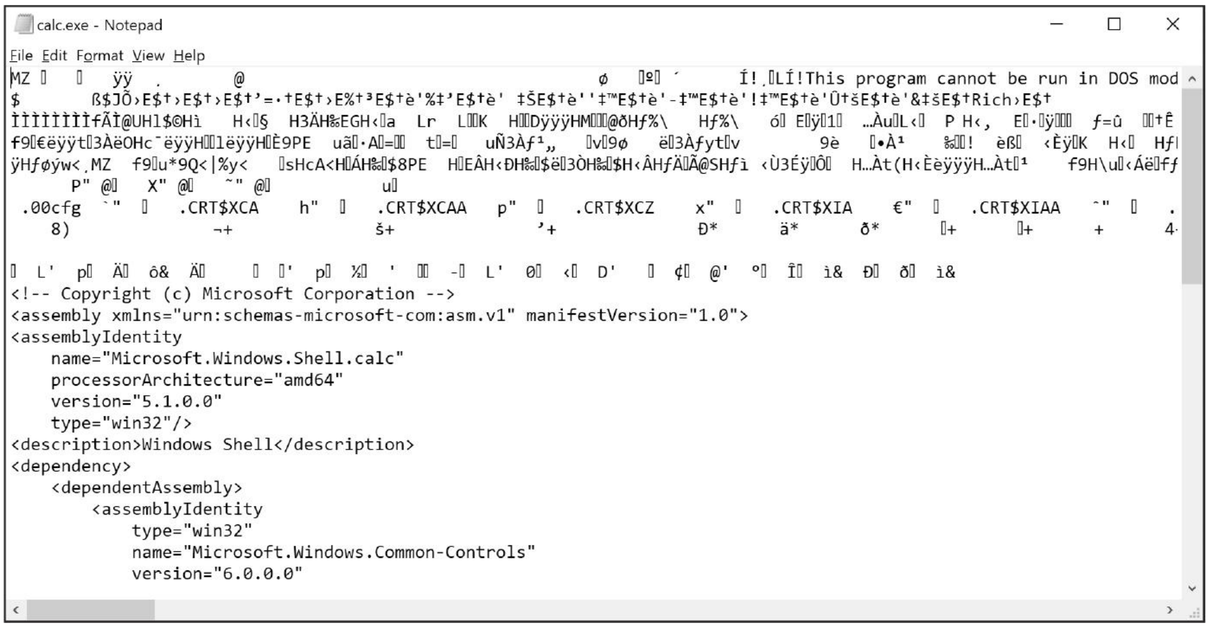
Oops! It looks like I forgot to plug in my flash drive.

The older `os.path` module can accomplish the same task with the `os.path.exists(`*`path`*`)`, `os.path.isfile(`*`path`*`)`, and `os.path.isdir(`*`path`*`)` functions, which act just like their `Path` function counterparts. As of Python 3.6, these functions can accept `Path` objects as well as strings of the filepaths.

The File Reading and Writing Process

Once you're comfortable working with folders and relative paths, you'll be able to specify the locations of files to read and write. The functions covered in the next few sections apply to plaintext files. *Plaintext files* contain only basic text characters and do not include font, size, or color information. Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files. You can open these with the Windows Notepad or macOS TextEdit application, and your programs can easily read their content, then treat it as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like that shown in Figure 10-5.



```
MZ         ÿÿ   .    @                                  ø    º  ´    Í! LÍ!This program cannot be run in DOS mod
$       ß$JÕ›E$†›E$†›E$†'=·†E$†›E%†³E$†è'%‡'E$†è'  ‡ŠE$†è''‡™E$†è'-‡™E$†è'!‡™E$†è'Û†šE$†è'&‡šE$†Rich›E$†
ÌÌÌÌÌÌÌÌfÃÌ@UH1$©Hì    H‹ §   H3ÄH‰EGH‹ a   Lr   L K   H DÿÿÿHM  @ðHƒ%\    Hƒ%\    ó  E ÿ 1    …Àu L‹    P H‹,   E ·ÿ    ƒ=û     †Ê
f9 €ëÿÿt 3ÀëOHc¯ëÿÿH  lëÿÿH È9PE   uã ·A =    t =    uÑ3Àƒ¹„    v 9ø    ë 3Àƒyt v         9è     •À¹     ‰ !   èß    ‹Èÿ K   H‹    Hƒ
ÿHƒøÿw<.MZ   f9 u*9Q<|%y<    sHcA<H ÁH‰ $8PE   H EÂH‹ÐH‰ $ë 3ÒH‰ $H‹ÂHƒÄ Ã@SHƒì  ‹Ù3Éÿ Ô    H…Àt(H‹ÈèÿÿÿH…Àt ¹     f9H\u ‹Áë fƒ
     P" @     X" @     ˜" @              u
 .00cfg   `"       .CRT$XCA     h"      .CRT$XCAA     p"      .CRT$XCZ      x"      .CRT$XIA      €"      .CRT$XIAA     ˆ"       .
    8)                 ¬+                 š+                ’+                Ð*          ä*          ð*          +           +          +          4

  L'   p   Ä   ô&   Ä          '   p   ½   '        -   L'   0   ‹   D'      ¢   @'   °   Î   i&   Ð   ð   i&
<!-- Copyright (c) Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
    name="Microsoft.Windows.Shell.calc"
    processorArchitecture="amd64"
    version="5.1.0.0"
    type="win32"/>
<description>Windows Shell</description>
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
```

Figure 10-5: The Windows calc.exe *program opened in Notepad*

Because we must handle each type of binary file in its own way, this book won't discuss reading and writing raw binary files directly.

Fortunately, many modules make working with binary files easier, and you'll explore one of them, the `shelve` module, later in this chapter.

The `pathlib` module's `read_text()` method returns the full contents of a text file as a string. Its `write_text()` method creates a new text file (or overwrites an existing one) with the string passed to it. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> p = Path('spam.txt')
>>> p.write_text('Hello, world!')
13
>>> p.read_text()
'Hello, world!'
```

These method calls create a *spam.txt* file with the content `'Hello, world!'`. The `13` that `write_text()` returns indicates that 13 characters were written to the file. (You can often disregard this return value.) The `read_text()` call reads and returns the contents of the new file as a string: `'Hello, world!'`.

Keep in mind that these `Path` object methods allow only basic interactions with files. The more common way of writing to a file involves using the `open()` function and file objects. There are three steps to reading or writing files in Python:

1. Call the `open()` function to return a `File` object.
2. Call the `read()` or `write()` method on the `File` object.
3. Close the file by calling the `close()` method on the `File` object.

We'll go over these steps in the following sections.

Note that as you begin working with files, you may find it helpful to be able to quickly see their extensions (*.txt*, *.pdf*, *.jpg*, and so on). Windows and macOS may hide file extensions by default, showing *spam.txt* as simply *spam*. To show extensions, open the settings for File Explorer (on Windows) or Finder (on macOS) and look for a checkbox that says something like "Show all filename extensions" or "Hide extensions for known file types." (The exact location and wording of this setting depend on the version of your operating system.)

Opening Files

To open a file with the `open()` function, pass it a string path indicating the file you want to open. This can be either an absolute path or a relative path. The `open()` function returns a `File` object.

Try this by creating a text file named *hello.txt* using Notepad or TextEdit. Enter **Hello, world!** as the content of this text file and save it in your user home folder. Then, enter the following in the interactive shell:

```
>>> from pathlib import Path
>>> hello_file = open(Path.home() /
'hello.txt', encoding='UTF-8')
```

The `open()` function will open the file in "reading plaintext" mode, or *read mode* for short. When a file is opened in read mode, Python lets you read the file's data but not write or modify it in any way. Read mode is the default mode for files you open in Python. But if you don't want to rely on Python's defaults, you can explicitly specify the mode by passing the string value `'r'` as a second argument to `open()`. For example, `open('/Users/Al/hello.txt', 'r')` does the same thing as `open('/Users/Al/hello.txt')`.

The `encoding` named parameter specifies what encoding to use when converting the bytes in the file to a Python text string. The correct encoding is almost always `'utf-8'`, which is also the default encoding used on macOS and Linux. However, Windows uses `'cp1252'` for its default encoding (also known as *extended ASCII*). This can cause problems when trying to read certain UTF-8 encoded text files with non-English characters on Windows, so it's a good habit to pass `encoding='utf-8'` to your `open()` function calls when opening files in plaintext read, write, or append mode. The binary read, write, and append modes don't use the `encoding` named parameter, so you can leave it out in those cases.

The call to `open()` returns a `File` object. A `File` object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you're already familiar with. In the previous example, you stored the `File` object in the variable `hello_file`. Now, whenever you want to read from or write to the file, you can do so by calling methods on the `File` object in `hello_file`.

Reading the Contents of Files

Now that you have a `File` object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the `File` object's `read()` method. Let's continue with the *hello.txt* `File` object you stored in `hello_file`. Enter the following into the interactive shell:

```
>>> hello_content = hello_file.read()
>>> hello_content
'Hello, world!'
```

You can think of the contents of a file as a single large string value; the `read()` method merely returns the string that is stored in the file.

Alternatively, you can use the `readlines()` method to get a *list* of string values from the file, one for each line of text. For example, create a file named *sonnet29.txt* in the same directory as *hello.txt* and place the following text in it:

```
When, in disgrace with fortune and men's
eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless
cries,
And look upon myself and curse my fate,
```

Make sure to separate the four lines with line breaks. Then, enter the following into the interactive shell:

```
>>> sonnet_file = open(Path.home() /
'sonnet29.txt', encoding='UTF-8')
>>> sonnet_file.readlines()
["When, in disgrace with fortune and men's
eyes,\n", 'I all alone beweep
my outcast state,\n', 'And trouble deaf
heaven with my bootless cries,\n',
'And look upon myself and curse my fate,']
```

Note that, except for the last line of the file, each of the string values ends with a newline character `\n`. A list of strings is often easier to work with than a single large string value.

Writing to Files

Python allows you to write content to a file, just as the `print()` function writes strings to the screen. You can't write to a file you've opened in read mode, though. Instead, you need to open it in "write plaintext" mode or "append plaintext" mode, called *write mode* and *append mode* for short.

Write mode will overwrite the existing file, which is similar to overwriting a variable's value with a new value. Pass `'w'` as the second argument to `open()` to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file. You can think of this mode as appending values to a list in a variable rather than overwriting the variable altogether. Pass `'a'` as the second argument to `open()` to open the file in append mode.

If the filename passed to `open()` does not exist, both write mode and append mode will create a new, blank file. After reading or writing a file, call the `close()` method before opening the file again.

Let's put these concepts together. Enter the following into the interactive shell:

```
>>> bacon_file = open('bacon.txt', 'w',
encoding='UTF-8')
>>> bacon_file.write('Hello, world!\n')
14
>>> bacon_file.close()
>>> bacon_file = open('bacon.txt', 'a',
encoding='UTF-8')
>>> bacon_file.write('Bacon is not a
vegetable.')
25
>>> bacon_file.close()
>>> bacon_file = open('bacon.txt',
encoding='UTF-8')
>>> content = bacon_file.read()
>>> bacon_file.close()
>>> print(content)
Hello, world!
Bacon is not a vegetable.
```

First, we open *bacon.txt* in write mode. As no *bacon.txt* file exists yet, Python creates one. Calling `write()` on the opened file and passing `write()` the string argument `'Hello, world!\n'` writes the string to the file and returns the number of characters written, including the newline. Then, we close the file.

To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write `'Bacon is not a vegetable.'` to the file and close it. Finally, to print the file contents to the screen, we open the file in its default read mode, call `read()`, store the resulting `File` object in `content`, close the file, and print `content`.

Note that the `write()` method does not automatically add a newline character to the end of the string like the `print()` function does. You will have to add this character yourself.

You can also pass a `Path` object to the `open()` function instead of the filename as a string.

Using with Statements

Every file on which your program calls `open()` needs `close()` called on it as well, but you may forget to include the `close()` function, or your program might skip over the `close()` call in certain circumstances.

Python's `with` statement makes it easier to automatically close files. A `with` statement creates something called a *context manager* that Python uses to manage resources. These resources, such as files, network connections, or segments of memory, often have setup and teardown steps during which the resource is allocated and later released so that other programs can make use of it. (Most of the time, however, you'll encounter `with` statements used to open files.)

The `with` statement adds a block of code that begins by allocating the resource and then releases it when the program execution leaves the block, which could happen due to a `return` statement, an unhandled exception being raised, or some other reason.

Here is typical code that writes and reads the content of a file:

```
file_obj = open('data.txt', 'w',
encoding='utf-8')
file_obj.write('Hello, world!')
file_obj.close()
file_obj = open('data.txt', encoding='utf-8')
```

```
content = file_obj.read()
file_obj.close()
```

Here is the equivalent code using a `with` statement:

```
with open('data.txt', 'w', encoding='UTF-8')
as file_obj:
    file_obj.write('Hello, world!')
with open('data.txt', encoding='UTF-8') as
file_obj:
    content = file_obj.read()
```

In the `with` statement example, notice that there are no calls to `close()` at all because the `with` statement automatically calls it when the program execution leaves the block. The `with` statement knows to do this based on the context manager it obtains from the `open()` function. Creating your own context managers is beyond the scope of this book, but you can learn about them from the online documentation at *https://docs.python.org/3/reference/datamodel.html#context-managers* or the book *Serious Python* by Julien Danjou (No Starch Press, 2018).

Saving Variables with the shelve Module

You can save variables in your Python programs to binary shelf files using the `shelve` module. This lets your program restore that data to the variables the next time it is run. You could use this technique to add Save and Open features to your program; for example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load the settings the next time it is run.

To practice using `shelve`, enter the following into the interactive shell:

```
>>> import shelve
>>> shelf_file = shelve.open('mydata')
>>> shelf_file['cats'] = ['Zophie', 'Pooka',
'Simon']
>>> shelf_file.close()
```

To read and write data using the `shelve` module, you first import `shelve`. Next, call `shelve.open()` and pass it a filename, then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary. When you're done, call `close()` on the shelf value. Here, our shelf value is stored in `shelf_file`. We create a list `cats` and write `shelf_file['cats'] = ['Zophie', 'Pooka', 'Simon']` to store the list in `shelf_file` as a value associated with the key `'cats'` (like in a dictionary). Then, we call `close()` on `shelf_file`.

After running the previous code on Windows, you should see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On macOS, you should see only a single *mydata.db* file, and Linux has a single *mydata* file. These binary files contain the data you stored in your shelf. The format of these binary files isn't important; you only need to know what the `shelve` module does, not how it does it. The module frees you from worrying about how to store your program's data to a file.

Your programs can use the `shelve` module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode; they allow both reading and writing once opened. Enter the following into the interactive shell:

```
>>> shelf_file = shelve.open('mydata')
>>> type(shelf_file)
<class 'shelve.DbfilenameShelf'>
>>> shelf_file['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelf_file.close()
```

Here, we open the shelf files to check that they stored the data correctly. Entering `shelf_file['cats']` returns the same list we created earlier. Now that we know the file stored the list correctly, we call `close()`.

Just like dictionaries, shelf values have `keys()` and `values()` methods that will return list-like values of the keys and values in the shelf. Because these return values are not true lists, you should pass them to the `list()` function to get them in list form. Enter the following into the interactive shell:

```
>>> shelf_file = shelve.open('mydata')
>>> list(shelf_file.keys())
['cats']
>>> list(shelf_file.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelf_file.close()
```

Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit, but if you want to save data from your Python programs, use the `shelve` module.

Project 4: Generate Random Quiz Files

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

- Creates 35 different quizzes
- Creates 50 multiple-choice questions for each quiz, in random order
- Provides the correct answer and three random wrong answers for each question, in random order
- Writes the quizzes to 35 text files
- Writes the answer keys to 35 text files

This means the code will need to do the following:

- Store the states and their capitals in a dictionary.
- Call `open()`, `write()`, and `close()` for the quiz and answer key text files.
- Use `random.shuffle()` to randomize the order of the questions and multiple-choice options.

Let’s get started.

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
  # randomQuizGenerator.py - Creates quizzes
  with questions and answers in
  # random order, along with the answer key

❶ import random

  # The quiz data. Keys are states and values
  are their capitals.
❷ capitals = {'Alabama': 'Montgomery',
  'Alaska': 'Juneau', 'Arizona':
  'Phoenix', 'Arkansas': 'Little Rock',
  'California': 'Sacramento', 'Colorado':
  'Denver', 'Connecticut': 'Hartford',
  'Delaware': 'Dover', 'Florida':
  'Tallahassee', 'Georgia': 'Atlanta',
  'Hawaii': 'Honolulu', 'Idaho': 'Boise',
  'Illinois': 'Springfield', 'Indiana':
  'Indianapolis', 'Iowa': 'Des Moines',
  'Kansas': 'Topeka', 'Kentucky': 'Frankfort',
  'Louisiana': 'Baton Rouge',
  'Maine': 'Augusta', 'Maryland': 'Annapolis',
  'Massachusetts': 'Boston',
  'Michigan': 'Lansing', 'Minnesota': 'Saint
  Paul', 'Mississippi': 'Jackson',
  'Missouri': 'Jefferson City', 'Montana':
  'Helena', 'Nebraska': 'Lincoln',
  'Nevada': 'Carson City', 'New Hampshire':
```

```
'Concord', 'New Jersey': 'Trenton',
'New Mexico': 'Santa Fe', 'New York':
'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio':
'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania':
'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota':
'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt
Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond',
'Washington': 'Olympia', 'West
Virginia':'Charleston', 'Wisconsin':
'Madison', 'Wyoming': 'Cheyenne'}

# Generate 35 quiz files.
❸ for quiz_num in range(35):
    # TODO: Create the quiz and answer key
files.

    # TODO: Write out the header for the
quiz.

    # TODO: Shuffle the order of the states.

    # TODO: Loop through all 50 states,
making a question for each.
```

Because this program will randomly order the questions and answers, you'll need to import the `random` module ❶ to make use of its functions. The `capitals` variable ❷ contains a dictionary with US states as keys and their capitals as values. And because you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with `TODO` comments for now) will go inside a `for` loop that loops 35 times ❸. (You can change this number to generate any number of quiz files.)

Step 2: Create the Quiz File

Now it's time to start filling in those `TODO`s.

The code in the loop will repeat 35 times, once for each quiz, so you have to worry about only one quiz at a time within the loop. First, you'll create the actual quiz file. It needs a unique filename and some kind of standard header, with places for the student to fill in a name, date, and class period. Then, you'll need to get a list of states in randomized order, which you can use later to create the questions and answers for the quiz.

Add the following lines of code to *randomQuizGenerator.py*:

```
# randomQuizGenerator.py - Creates quizzes
with questions and answers in
# random order, along with the answer key

--snip--

# Generate 35 quiz files.
for quiz_num in range(35):
    # Create the quiz and answer key files.
    quiz_file = open(f'capitalsquiz{quiz_num
+ 1}.txt', 'w', encoding='UTF-8') ❶
    answer_file =
open(f'capitalsquiz_answers{quiz_num +
1}.txt', 'w', encoding='UTF-8') ❷

    # Write out the header for the quiz.
    quiz_file.write('Name:\n\nDate:
\n\nPeriod:\n\n') ❸
    quiz_file.write((' ' * 20) + f'State
Capitals Quiz (Form{quiz_num + 1})')
    quiz_file.write('\n\n')

    # Shuffle the order of the states.
    states = list(capitals.keys())
    random.shuffle(states) ❹

    # TODO: Loop through all 50 states,
making a question for each.
```

The quizzes will use the filenames *capitalsquiz<N>.txt*, where *<N>* is a unique number that comes from `quiz_num`, the `for` loop's counter. We'll store the answer key for *capitalsquiz<N>.txt* in the text files named *capitalsquiz_answers<N>.txt*. On each iteration of the loop, the code will replace the `{quiz _num + 1}` placeholders in these filenames with a unique number. For example, it names the first quiz and answer key *capitalsquiz1.txt* and *capitalsquiz _answers1.txt*. We create these files with calls to the `open()` function at ❶ and ❷, passing `'w'` as the second argument to open them in write mode.

The `write()` statements at ❸ create a quiz header for the student to fill out. Finally, we generate a randomized list of US states with the help of the `random.shuffle()` function ❹, which randomly reorders the values in any list that is passed to it.

Step 3: Create the Answer Options

Now you need to generate answer options A to D for each question using another `for` loop. Later, a third, nested `for` loop will write these multiple-choice options to the files. Make your code look like the following:

```
# randomQuizGenerator.py - Creates quizzes
with questions and answers in
# random order, along with the answer key

--snip--

    # Loop through all 50 states, making a
question for each.
    for num in range(50):

        # Get right and wrong answers.
        correct_answer =
capitals[states[num]]
        wrong_answers =
list(capitals.values())
        del
wrong_answers[wrong_answers.index(correct_ans
wer)]
        wrong_answers =
random.sample(wrong_answers, 3)
        answer_options = wrong_answers +
[correct_answer]
        random.shuffle(answer_options)

        # TODO: Write the question and answer
options to the quiz file.

        # TODO: Write the answer key to a
file.
```

The correct answer is easy to create; it's already stored as a value in the `capitals` dictionary. This loop will iterate through the states in the shuffled `states` list, find each state in `capitals`, and store that state's corresponding capital in `correct_answer`.

Creating the list of possible wrong answers is trickier. You can get it by duplicating the values in the `capitals` dictionary, deleting the correct answer, and selecting three random values from this list. The `random.sample()` function makes performing this selection easy. Its first argument is the list you want to select from, and the second argument is the number of values to select. The full list of answer options combines these three wrong answers with the correct answers. Finally, we randomize the answers so that the correct response isn't always choice D.

Step 4: Write the Content to the Files

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

```
# randomQuizGenerator.py - Creates quizzes
with questions and answers in
# random order, along with the answer key

--snip--

    # Loop through all 50 states, making a
question for each.
    for num in range(50):
        --snip--

        # Write the question and the answer
options to the quiz file.
        quiz_file.write(f'{num + 1}. Capital
of {states[num]}:\n')
       ❶ for i in range(4):
           ❷ quiz_file.write(f"
{'ABCD'[i]}. {answer_options[i]}\n")
        quiz_file.write('\n')

        # Write the answer key to a file.
       ❸ answer_file.write(f"{num + 1}.
{'ABCD'[answer_options.index(correct_answer)]
}")
    quiz_file.close()
    answer_file.close()
```

A `for` loop iterates through integers `0` to `3` to write the answer options in the `answer_options` list to the file ❶. The expression `'ABCD'[i]` at ❷ treats the string `'ABCD'` as an array and will evaluate to `'A'`,`'B'`, `'C'`, and `'D'` on each respective iteration through the loop.

In the final line of the loop, the expression `answer_options.index(correct_answer)` ❸ will find the integer index of the correct answer in the randomly ordered answer options, causing the correct answer's letter to be written to the answer key file.

After you run the program, your *capitalsquiz1.txt* file should look something like this. Of course, your questions and answer options will depend on the outcome of your `random.shuffle()` calls:

```
Name:

Date:

Period:

                    State Capitals Quiz (Form
1)

1. What is the capital of West Virginia?
    A. Hartford
    B. Santa Fe
    C. Harrisburg
    D. Charleston

2. What is the capital of Colorado?
    A. Raleigh
    B. Harrisburg
    C. Denver
    D. Lincoln

--snip--
```

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

```
1. D
2. C
```

```
--snip--
```

Randomly ordering the question set and corresponding answer key by hand would take hours to do, but with a little bit of programming knowledge, you can automate this boring task for not just a state capitals quiz but any multiple-choice exam.

Summary

Operating systems organize files into folders (also called directories), and use paths to describe their locations. Every program running on your computer has a current working directory, which allows you to specify filepaths relative to the current location instead of entering the full (or absolute) path. The `pathlib` and `os.path` modules have many functions for manipulating filepaths.

Your programs can also directly interact with the contents of text files. The `open()` function can open these files to read in their contents as one large string (with the `read()` method) or as a list of strings (with the `readlines()` method). The `open()` function can also open files in write or append mode to create new text files or add to existing text files, respectively.

In previous chapters, you used the clipboard as a way of getting large amounts of text into a program, rather than typing it directly. Now you can have your programs read files from the hard drive, which is a big improvement, as files are much less volatile than the clipboard.

In the next chapter, you will learn how to handle the files themselves by copying them, deleting them, renaming them, moving them, and more.

Practice Questions

1. What is a relative path relative to?
2. What does an absolute path start with?
3. What does `Path('C:/Users') / 'Al'` evaluate to on Windows?
4. What does `'C:/Users' / 'Al'` evaluate to on Windows?
5. What do the `os.getcwd()` and `os.chdir()` functions do?
6. What are the . and .. folders?
7. In *C:\bacon\eggs\spam.txt*, which part is the directory name, and which part is the base name?
8. What three “mode” arguments can you pass to the `open()` function for plaintext files?

9. What happens if an existing file is opened in write mode?
10. What is the difference between the `read()` and `readlines()` methods?
11. What data structure does a shelf value resemble?

Practice Programs

For practice, design and write the following programs.

Mad Libs

Create a Mad Libs program that reads in text files and lets the user add their own text anywhere the word *ADJECTIVE*, *NOUN*, *ADVERB*, or *VERB* appears in the text file. For example, a text file may look like this:

```
The ADJECTIVE panda walked to the NOUN and
then VERB. A nearby NOUN was
unaffected by these events.
```

The program would find these occurrences and prompt the user to replace them:

```
Enter an adjective:
silly
Enter a noun:
chandelier
Enter a verb:
screamed
Enter a noun:
pickup truck
```

It would then create the following text file:

```
The silly panda walked to the chandelier and
then screamed. A nearby
pickup truck was unaffected by these events.
```

The program should print the results to the screen in addition to saving them to a new text file.

Regex Search

Write a program that opens all *.txt* files in a folder and searches for any line that matches a user-supplied regular expression, then prints the results to the screen.