Cahpter-2: Reading and Writing Files

3.15 Files and File Paths

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 with filename project.docx is 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).

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 /.

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.
This is simple to do with the Path() function in the pathlib module. If you pass it the string values of individual file and folder names in your path, Path() will return a string with a file path using the correct path separators.

In [ ]:

```
# Example Path()
from pathlib import Path
Path('spam', 'bacon', 'eggs')
```

To get a simple text string of this path, we can pass it to the str() function, which returns 'spam\bacon\eggs'.

In [ ]:

```
str(Path('spam', 'bacon', 'eggs'))
```

The / operator that is normally used for division can also be used to combine Path objects and strings. This is helpful for modifying a Path object after it is already created it with the Path() function.

In [ ]:

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

In [ ]:

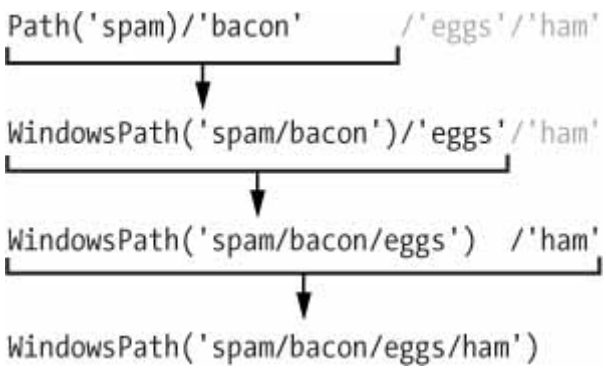
```
Path('spam') / Path('bacon/eggs')
```

In [ ]:

```
Path('spam') / Path('bacon', 'eggs')
```

Python evaluates the / operator from left to right and evaluates to a Path object, so either the first or second

leftmost value must be a Path object for the entire expression to evaluate to a Path object.



3.15.1 The Current Working Directory

Every program that runs on your computer has a current working directory, or cwd. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.
The current working directory can be obtained as a string value with the Path.cwd() function and change it using os.chdir().

In [ ]:

```
from pathlib import Path
import os
Path.cwd()
```

In [ ]:

```
os.chdir('C:\\Windows\\System32')
Path.cwd()
```

3.15.2 The Home Directory

All users have a folder for their own files on the computer called the home directory or home folder. You can get a Path object of the home folder by calling Path.home(). The home directories are located in a set place depending on your operating system:
On Windows, home directories are under C:\Users.
On Mac, home directories are under /Users.
On Linux, home directories are often under /home.

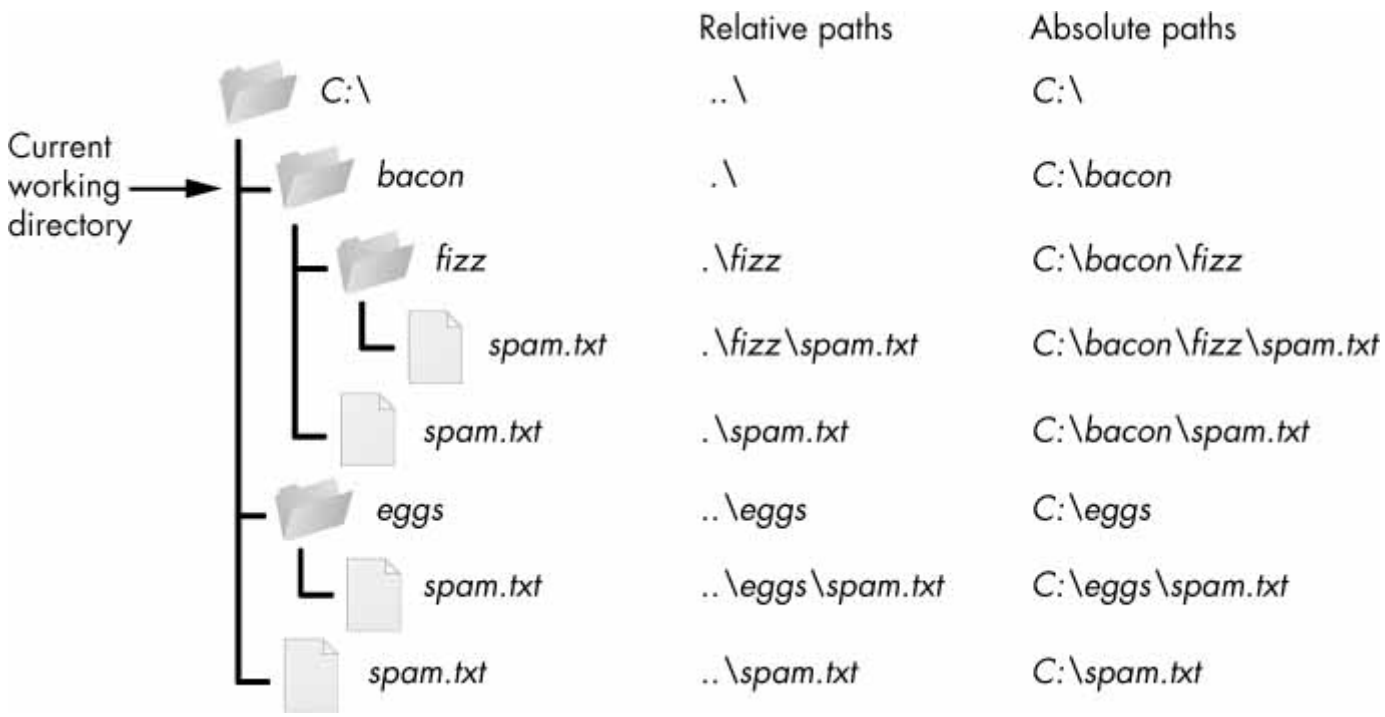
In [ ]:

```
Path.home()
```

3.15.3 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 current working directory

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



3.15.4 Creating New Folders Using the os.makedirs() Function

Programs can create new folders (directories) with the os.makedirs() function.\ Example below not just creates the C:\delicious folder but also a walnut folder inside C:\delicious and a waffles folder inside C:\delicious\walnut.

In [ ]:

```
import os
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:

In [ ]:

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

3.15.4 Handling Absolute and Relative Paths \

The pathlib module provides methods for checking whether a given path is an absolute path and returning the absolute path of a relative path.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.

In [ ]:

```
Path.cwd().is_absolute()
```

In [ ]:

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

To get an absolute path from a relative path, you can put Path.cwd() / in front of the relative Path object.

In [ ]:

```
Path.cwd() / Path('my/relative/path')
```

3.16 The OS path Module

The os.path module also has some useful functions related to absolute and relative paths:
Calling os.path.abspath(path) will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
Calling os.path.isabs(path) will return True if the argument is an absolute path and False if it is a relative path.
Calling os.path.relpath(path, start) will return a string of a relative path from the start path to path. If start is not provided, the current working directory is used as the start path.

In [ ]:

```
os.path.abspath('.')
```

In [ ]:

```
os.path.abspath('.\\Scripts')
```

In [ ]:

```
os.path.isabs('.')
```

In [ ]:

```
os.path.isabs(os.path.abspath('.'))
```

In [ ]:

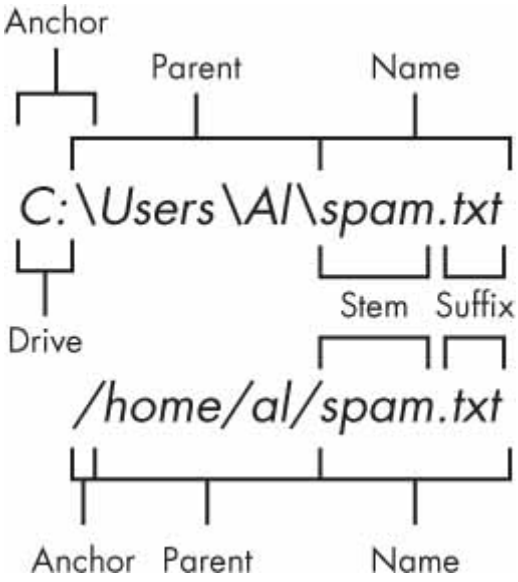
```
os.path.relpath('C:\\Windows', 'C:\\')
```

In [ ]:

```
os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
```

3.16.1 Getting the Parts of a File Path \

Given a Path object, you can extract the file path's different parts as strings using several Path object attributes.



The parts of a file path 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.

In [ ]:

```
p = Path('C:/Users/Al/spam.txt')
p.anchor
```

In [ ]:

```
p.parent # This is a Path object, not a string.
```

In [ ]:

```
p.name
```

In [ ]:

```
 p.stem
```

In [ ]:

```
p.suffix
```

In [ ]:

```
p.drive
```

The parents attribute (which is different from the parent attribute) evaluates to the ancestor folders of a Path

object with an integer index:

In [ ]:

```
Path.cwd()
```

In [ ]:

```
Path.cwd().parents[0]
```

In [ ]:

```
Path.cwd().parents[1]
```

The older os.path module also has similar functions for getting the different parts of a path written in a string value. Calling os.path.dirname(path) will return a string of everything that comes before the last slash in the path argument. Calling os.path.basename(path) will return a string of everything that comes after the last slash in the path argument. The directory (or dir) name and base name of a path are outlined in Figure

In [ ]:

```
# Example
calcFilePath = 'C:\\Windows\\System32\\calc.exe'
os.path.basename(calcFilePath)
```

In [ ]:

```
os.path.dirname(calcFilePath)
```

In [ ]:

```
#use split to get both base name and directory name as follows:
calcFilePath = 'C:\\Windows\\System32\\calc.exe'
os.path.split(calcFilePath)
```

In [ ]:

```
#  os.sep variable is set to the correct folder-separating slash for the computer running t
calcFilePath.split(os.sep)
```

3.16.2 Finding File Sizes and Folder Contents

The os.path module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.\

- Calling os.path.getsize(path) will return the size in bytes of the file in the path argument.\
- Calling os.listdir(path) will return a list of filename strings for each file in the path argument.\

In [ ]:

```
os.path.getsize('C:\\Windows\\System32\\calc.exe')
```

In [ ]:

```
os.listdir('C:\\Windows\\System32')
```

In [ ]:

```
# finds the total size of all files in the given directory.
totalSize = 0
for filename in os.listdir('C:\\Windows\\System32'):
      totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32', filenam
print(totalSize)
```

3.16.3Modifying a List of Files Using Glob Patterns

Path objects have a glob() method for listing the contents of a folder according to a glob pattern. Glob patterns are like a simplified form of regular expressions often used in command line commands.
The glob() method returns a generator object that you'll need to pass to list() to easily view in theshell:

In [ ]:

```
p = Path('C:\\Users\\91984\\Documents\\python')
p.glob('*')
```

In [ ]:

```
 list(p.glob('*'))#The asterisk (*) stands for "multiple of any characters," so p.glob('*')
```

In [ ]:

```
 # Lists all text files.
list(p.glob('*.txt'))
```

In [ ]:

```
#The glob expression 'project?.docx' will return 'project1.docx' or 'project5.docx', but it
list(p.glob('project?.docx'))
```

In [ ]:

```
# The glob expression '*.?x?' will return files with any name and any three-character exten
list(p.glob('*.?x?'))
```

3.16.4 Checking Path Validity

Path objects have methods to check whether a given path exists and whether it is a file or folder.\

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

In [ ]:

```
winDir = Path('C:/Windows')
notExistsDir = Path('C:/This/Folder/Does/Not/Exist')
calcFile = Path('C:/Windows/System32/calc.exe')
print(winDir.exists())
print(winDir.is_dir())
print(notExistsDir.exists())
print(calcFile.is_file())
```

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.

3.17 The File Reading/Writing Process

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.\ Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs.

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

In [ ]:

```
from pathlib import Path
p = Path('C:\\Users\\91984\\Documents\\python\\hello.txt')
p.write_text('Hello, world!')
p.read_text()
```

The 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.\

3.17.1 Opening Files with the open() Function

In [ ]:

```
# to open file using open() function pass it a string path of file to be opened
hellofile=open(Path.home()/'hello.txt')
```

In [ ]:

```
#open function can also accept strings
helloFile = open('C:\\Users\\91984\\hello.txt')
```

3.17.2 Reading the Contents of Files

To read the entire contents of a file as a string value, use the File object's read() method.

In [ ]:

```
helloContent = helloFile.read()
helloContent
```

In [ ]:

```
 # The readlines() method can be used to gett a list of string values from the file, one st
helloContent = helloFile.readlines()
helloContent
```

3.17.3 Writing to Files

To write content to a file open it in "write plaintext" mode or "append plaintext" mode, or write mode and append mode for short.
Write mode will overwrite the existing file and start from scratch.Pass 'w' as the second argument to open() to open the file in write mode.
Append mode, will append text to the end of the existing file.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 and append mode will create a new, blank file.
After reading or writing a file, call the close() method before opening the file again.

In [ ]:

```
baconFile = open('bacon.txt', 'w')
baconFile.write('Hello, world!\n')
baconFile.close()
```

In [ ]:

```
baconFile = open('bacon.txt', 'a')
baconFile.write('Bacon is not a vegetable.')
```

In [ ]:

```
baconFile = open('bacon.txt')
content = baconFile.read()
baconFile.close()
print(content)
```

3.18 Saving Variables with the shelve Module

The shelve module can be used to save variables in Python programs to a binary shelf files ,This helps the program to restore data to variables from the hard drive.The shelve module will let you add Save and Open features to your program.

In [2]:

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

After running the above code on Windows,three new files in the current working directory will be created: mydata.bak, mydata.dat, and mydata.dir. On macOS, only a single mydata.db file will be created,which stores these binary files containing the data stored in shelf.

In [3]:

```
shelfFile = shelve.open('mydata')
type(shelfFile)
```

Out[3]:

```
shelve.DbfilenameShelf
```

In [4]:

```
shelfFile['cats']
```

Out[4]:

```
['Zophie', 'Pooka', 'Simon']
```

In [5]:

```
shelfFile.close()
```

In [8]:

```
shelfFile = shelve.open('mydata')
list(shelfFile.keys()) # returns key of the list
```

Out[8]:

```
['cats']
```

In [9]:

```
list(shelfFile.values())# returns values associated with key of the list
```

Out[9]:

```
[['Zophie', 'Pooka', 'Simon']]
```

In [10]:

```
shelfFile.close()
```

3.19 Saving Variables with the pprint.pformat() Function

Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use.
Using pprint.pformat() will give you a string that you can write to a .py file.
This file will be your very own module that you can import whenever you want to use the variable stored in it.

In [11]:

```
import pprint
cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
pprint.pformat(cats)
```

Out[11]:

```
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pook
a'}]"
```

In [14]:

```
fileObj = open('myCats.py', 'w')
fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
```

Out[14]:

```
83
```

In [15]:

```
fileObj.close()
```

In [18]:

```
#Python programs can even generate other Python programs.
import myCats
myCats.cats
```

Out[18]:

```
[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]
```

In [17]:

```
 myCats.cats[0]
```

Out[17]:

```
{'desc': 'chubby', 'name': 'Zophie'}
```

In [19]:

```
myCats.cats[0]['name']
```

Out[19]:

```
'Zophie'
```

3.20 Project: Generating Random Quiz Files

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

The code will need to do the following:

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

In [ ]:

```

```