ATBS: Chapter 8 notes
Reading and Writing Files
https://automatetheboringstuff.com/chapter8/

* Note: a lot of these files paths are written from the perspective of a Windows user, which is why they might look weird.

*Also note that a lot of the notes for this chapter are copied straight from the book because these concepts are mostly review / reference.

Backslash on Windows and Forward Slash on OS X and Linux
Forward slash (/) is used as a path separator in MacOS and Linux, backslash (\) is used in Windows. **You must account for both with Python.**

-Both can be accounted for using the os.path.join() function

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

“I’m running these interactive shell examples on Windows, so os.path.join('usr', 'bin', 'spam') returned 'usr\\bin\\spam'. (Notice that the backslashes are doubled because each backslash needs to be escaped by another backslash character.) If I had called this function on OS X or Linux, the string would have been 'usr/bin/spam'.”

“The os.path.join() function is helpful if you need to create strings for filenames. These strings will be passed to several of the file-related functions introduced in this chapter. For example, the following example joins names from a list of filenames to the end of a folder’s name:”

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

The Current Working Directory
Every program has a CWD. You can get the CWD as a string value with the os.getcwd() function.

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

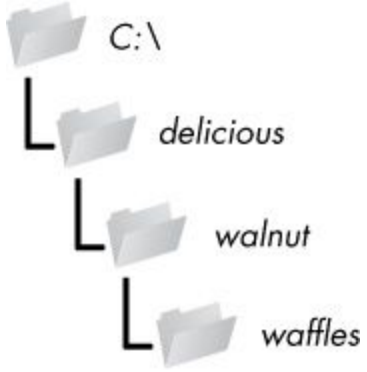
Absolute vs. relative paths

Two ways to specify a path:
-Absolute (path begins with root folder)
-Relative (relative to program's current CWD)

*See figure 8-2 in book for graph

Creating new folders with os.makedirs()

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



The os.path module

The os.path module contains many helpful functions related to filenames and file paths (i.e. os.path.join()

See full doc: http://docs.python.org/3/library/os.path.html.

Handling 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.”

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd() 'C:\\Python34'
```

If you need a path’s dir name and base name together, you can just call os.path.split() to get a tuple value with these two strings, like so:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

“Also, note that os.path.split() does not take a file path and return a list of strings of each folder. For that, use the split() string method and split on the string in os.sep. Recall from earlier that the os.sep variable is set to the correct folder-separating slash for the computer running the program.”

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

Finding file sizes and folder contents

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. (Note that this function is in the os module, not os.path.)

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

If you want to find the total size of all files in this directory, you can run os.path.getsize() and os.listdir() together

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
      totalSize = totalSize +
os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(totalSize)
1117846456
```

Checking path validity

Os.path module provides functions to check whether a given path exists/whether it is a file or folder.

Calling **os.path.exists(path**) will return True if the file or folder referred to in the argument exists and will return False if it does not exist.

Calling **os.path.isfile(path)** will return True if the path argument exists and is a file and will return False otherwise.

Calling **os.path.isdir(path)** will return True if the path argument exists and is a folder and will return False otherwise.

```
>>> os.path.exists('C:\\Windows')
True
```

```
>>> os.path.exists('C:\\some_made_up_folder')
False
```

The file reading/writing process

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.

Opening files with the open() function

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

open() does basically exactly what you think that it would.

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

Reading the contents of files

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

readlines() gives you a list of string values from the file

Writing files

“Write mode will overwrite the existing file and start from scratch, just like when you overwrite 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 as appending 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.”**

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

Saving variables with the shelve module

“You can save variables in your Python programs to binary shelf files using the shelve module. This way, your program can restore data to variables from the hard drive. The shelve module will let you 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 them the next time it is run.”

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

After running the previous code, you will see a single mydata.db file will be created.

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

```
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.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. Since these methods return list-like values instead of true lists, you should pass them to the list() function to get them in list form. Enter the following into the interactive shell:”

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

Saving Variables with the pprint.pformat() Function

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc':
'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name':
'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Here, we import pprint to let us use pprint.pformat(). We have a list of dictionaries, stored in a variable cats. To keep the list in cats available even after we close the shell, we use pprint.pformat() to return it as a string. Once we have the data in cats as a string, it’s easy to write the string to a file, which we’ll call myCats.py.

The modules that an import statement imports are themselves just Python scripts. When the string from pprint.pformat() is saved to a .py file, the file is a module that can be imported just like any other.

And since Python scripts are themselves just text files with the .py file extension, your Python programs can even generate other Python programs. You can then import these files into scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

“The benefit of creating a .py file (as opposed to saving variables with the shelve module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor. For most applications, however, saving data using the shelve module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. File objects, for example, cannot be encoded as text.