MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

COMP7004 - Systems Scripting

Lecture 14: Reading and Writing Files

Dr. Vincent Emeakaroha
vincent.emeakaroha@mtu.ie

Semester 2, 2025

www.mtu.ie

Overview

- Variables are fine way to store data while your program is running.
  - For example, we will write a birthday program with dictionary data type later.
- However, if you want your data to persist even after your program has finished, you need to save it to a file.
- A file content can be thought of as a single string value that is potentially large in size.

Files and File Paths

- A file has two key properties:
  - a filename (usually written as one word), and
  - a path (which specifies the location of a file on a computer).
- The part of the filename after the last period is called the file's extension and it denotes a file type.
  - Example: ".py"
- Files are contained in folders known as directories. Folders can contain files and other folders. The top-most folder is known as the root folder, which contains other folders.
- While folder is the more modern name for directory, note that current working directory is the standard term, not current working folder.

Backslash on Windows and Forward Slash on OS X and Linux

- On Windows, paths are written using backslashes (\) as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator.
- If you want your program to work on both operating systems, you will have to write your Python scripts to handle both cases.
- Fortunately, this is simple to do with the os.path.join() function.
  - If you pass it the string value of individual files and folder names in your path, it will return a string with a file path using the correct path separators.
  - An example follows.

Backslash on Windows and Forward Slash on OS X and Linux

```
>>> import os
>>> os.path.join(os.sep, 'usr', 'local', 'bin', 'myScript')
'/usr/local/bin/myScript'
```

- If you run this on Windows, (assuming the directory structure exist in Windows), the output will be different. It will output something like:
  - '\\usr\\local\\bin\\myScript' (The double backslash is because each backslash need to be escaped with another backslash).
  - The os.sep provide dynamic separator to make path valid.
- The os.path.join() function is very handy if you need to create filenames by combining strings.

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 following two functions manipulates the cwd:
  - os.getcwd() returns the current working directory as a string value.
  - os.chdir() changes the current working directory to the one passed as a string argument to the function. If the directory does not exist, it will return an error.

```
>>> import os
>>> os.getcwd()
'/Users/chima/Documents'
>>> os.chdir('/Users/chima/Documents/backup')
>>> os.getcwd()
'/Users/chima/Documents/backup'
```

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 a shorthand for "this directory". Two periods ("dot-dot") means "the parent folder".

Handling Absolute and Relative Paths

- The os.path module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.
  - os.path.abspath(*path*) returns a string of the absolute path of the argument. This is an easy way to convert a relative path to an absolute one.
  - os.path.isabs(*path*) returns True if the argument is an absolute path and False if it is a relative path.
  - os.path.relpath(*path*, *start*) returns 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 starting point.

Handling Absolute and Relative Paths

- 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.
- Example:

```
>>> import os
>>> path = '/usr/local/bin/myScript.sh'
>>> os.path.basename(path)
'myScript.sh'
>>> os.path.dirname(path)
'/usr/local/bin'
```

- If you need a path's *dirname* and *basename* together, you can just call os.path.split() to get a tuple value with these two strings.

```
>>> os.path.split(path)
('/usr/local/bin', 'myScript.sh')
```

Creating New Folders

- One can create new folders (directories) in a Python program with the os.makedirs() function.
- Example:

```
>>> import os
>>> os.makedirs("/usr/local/chima")
```

- This will not just create “/usr” folder but also a “local” folder inside ”/usr” and a “chima” folder inside “/usr/local”. That is, this function will create any intermediate folder in order to ensure that the full path exists.
- The interoperable and recommended way to do this from a root folder is as follows:

```
>>> import os
>>> os.makedirs(os.path.join(os.sep, "usr", "local", "chima"))
```

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 the 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('/Users/chima/Documents/test/msg.txt')
3096
>>> os.listdir('/Users/chima/Documents/test')
['info.txt', 'msg.txt']
```

- If you want to find the total size of all the file in this directory, you can use os.path.getsize() and os.listdir() together. See following example.

Calculating Total Size of Files in a Folder

```
>>> totalsize = 0
>>> for filename in os.listdir('/Users/chima/Documents/test'):
        totalsize = totalsize + os.path.getsize(os.path.join('/Users/chima/Documents/test', filename))

>>> print(totalsize)
4644
```

- The program above loops through the files in the folder and add their sizes together. To get the size, the path has to be joined with the string name to form a real file.
- Note how multiple functions were combined together in this example program.

Checking Path Validity

- Many Python functions will crash with an error if supplied with a path that does not exist. The os.path module provides functions to check whether a given path exist and 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 exist and otherwise returns False.
  - Calling os.path.isfile(*path*) will return True if the path argument exist and is a file and otherwise returns False.
  - Calling os.path.isdir(*path*) will return True if the path argument exist and is a folder and otherwise returns False.

Checking Path Validity Example

```
>>> os.path.exists('/Users/chima/Documents')
True
>>> os.path.exists('/how/some_makeup_folder/')
False
>>> os.path.isdir('/Users/chima/Documents/test')
True
>>> os.path.isfile('/Users/chima/Documents/test')
False
>>> os.path.isdir('/Users/chima/Documents/test/info.txt')
False
>>> os.path.isfile('/Users/chima/Documents/test/info.txt')
True
```

- If I want to check for a flash drive with the volume name D:\ on my Window computer, I could do that with the following:
  - os.path.exists('D:\\')

File Reading / Writing Process

- The first set of functions we will consider applies to plaintext files.
- Plaintext file contains only basic text characters and do not include font, size, or colour information. Text files with the .txt extension or Python script files with .py extension are example of plaintext files.
- Binary files are all other file types such as word processing document, PDF, images, and executable programs. Every binary file must be handled in its own way. However, Python provides many modules that make working with binary files easy.
  - shelve module is an example that we will see later.

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 File with the open() Function

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

```
>>> helloFile = open('/Users/chima/Documents/test/hello.txt')
```

- The above command will open the file in "reading plaintext" mode or read mode for short. When a file is opened in read mode, you can only read data from the file, you cannot write or modify it in anyway.

Opening File with the open() Function

- The read mode is the default mode for files you open in Python.
- Without relying on Python's default behaviour, you can explicitly specify the mode by passing the string value 'r' as a second argument to the function open().
  - So open('/Users/chima/Documents/test/hello.txt', 'r') and open('/Users/chima/Documents/test/hello.txt') do the same thing.
- The call to open() returns a File object. A File object represents a file on your computer. It is another type of value in Python.
- In the previous example, we stored the File object in the variable helloFile. Now, whenever you want to read from the file, you can do so by calling methods on the File object stored in helloFile.

Reading the Contents of Files

- If you want to read the entire content of a file as a string value, use the File object's read() method. Let us continue with the hello.txt object we stored in helloFile.

```
>>> helloFile = open('/Users/chima/Documents/test/hello.txt')

>>> helloContent = helloFile.read()
>>> helloContent
'Hello World\n'
```

- So in the above example, the read() method returns the string that is stored in the file as a single string.

The readlines() Method

- Alternatively, you can use the readlines() method to get a list of string values from the file, one string for each line of text.
- Sample file content:

```
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 course my fate,
```

- Example code snippet:

```
>>> sonnetFile = open('sonnet.txt')
>>> sonnetFile.readlines()
```

- Output:

```
["When, in disgrace with fortune and men's eyes,\n", 'I all alone beweep my outcast state,\n', 'And trouble de
af heaven with my bootless cries,\n', 'And look upon myself and course my fate,\n']
```

Writing to Files

- Python allows you to write content to a file in a way similar to how the print() function writes strings to the screen.
- To write to a file, it must be opened in a “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, just like when you overwrite a variable value with a new value. Pass ‘w’ as a second argument to open() to open a file in write mode.
- Append mode on the other hand will append text to the end of existing file. It does not overwrite the existing content. Pass ‘a’ as second argument to open() to open the file in append mode.

Writing to Files

- If the filename passed to open() does not exist in both write and append modes, a new blank file will be created and opened for writing or appending respectively.
- After reading or writing to a file, call the close() method before opening the file again.
- Example follows on next slide.

Writing to Files

```
>>> 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')
24
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable
```

- The write() method does not automatically add a newline at the end of a string just like the print() function does. You will have to add the character yourself.
- Note that we did not specify the full path of the file "bacon.txt" because it is located in the current working directory.

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 enables you to add save and open features to your program. See following example:

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

- One can make changes to the shelf value as if it were a dictionary.
- After running this program on OS X or Linux a single 'myData.db' will be created for storing the values. On Windows, three files will be created for this purpose.

Saving Variables with the Shelve Module

- We can use the shelve module to later reopen and retrieve data from the shelf files. Shelf values do not have to be opened in read or write mode - they can do both once opened. See following example:

```
>>> 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.

Handling Command Line Arguments

- The sys module provides access to variables used or maintained by Python.
- One of the options to handle command line argument is through the use of sys.argv variable.
  - It is a simple list structure.
  - len(sys.argv) provides the number of command line arguments including the name of script.
  - sys.argv[0] is the name of the current Python script.
- Example
  - commandLine.py

commandLine.py

```
# A sample command line Python script

import sys

# Get the length of list meaning the number of command line arguments
num = len(sys.argv)
print("The number of arguments are: ", num)

# Get the name of script
script = sys.argv[0]
print("The name of script file is: ", script)

# Get first, second arguments provided
print("First argument: ", sys.argv[1], " Second argument: ", sys.argv[2])

# Go through the list and output all
for var in sys.argv:
    print(var)
```

- Note that len(sys.argv) includes the name of script at sys.argv[0].
  - This must be subtracted from len function to get the actual number of arguments provided to the script.

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University