IT Scripting and Automation

Additional OS interactions
and
Interacting with Data
in Python

Lecturer: Art Ó Coileáin

Using the *os* Module to Interact with Data

- In a previous section we looked a some basic interactions with the Bash shell in Python using ***subprocess calls*** using the ***subprocess*** module.

- In this section we will look at the:
  - **os**
  - **shutil**
  - **filecmp**
  - **dircmp**
  - **hashlib**

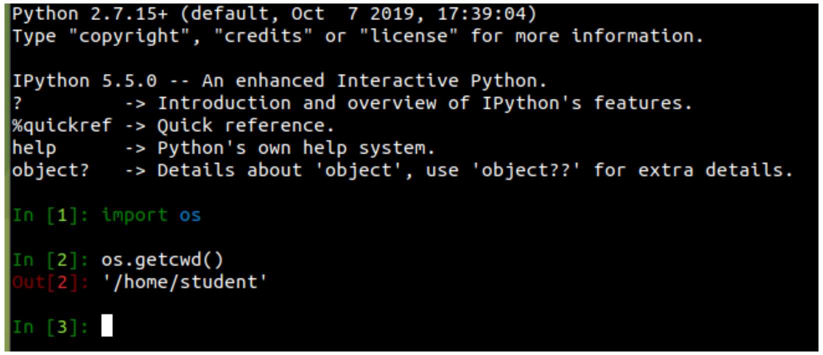
  Python Modules.

- Starting with the **os** module, it is a portable application programming interface (API) to system services. The ***OS module*** contains over 200 methods, and many of those methods deal with data.

- Starting on the next page, we will look at examples of using the OS module to create, list, rename and remove directories.

Using the *os* Module to Interact with Data

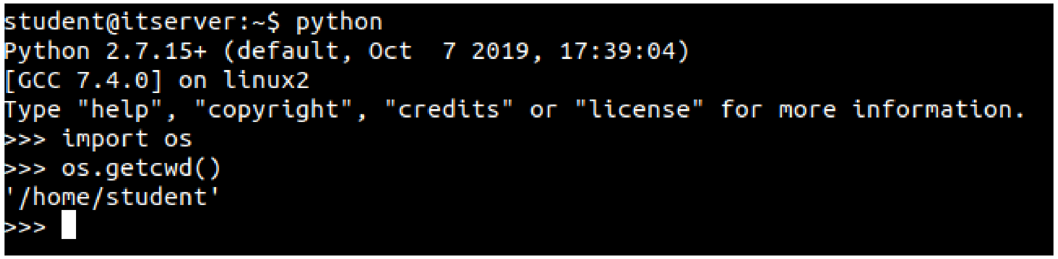
The following example of using the **OS** module will work with iPython or with the Python cmd line environment (*both Python ver. 2.7.15 here*):



```
Python 2.7.15+ (default, Oct  7 2019, 17:39:04)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import os

In [2]: os.getcwd()
Out[2]: '/home/student'

In [3]: 
```

iPython

Python 2.7.x environment



```
student@itserver:~$ python
Python 2.7.15+ (default, Oct  7 2019, 17:39:04)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.getcwd()
'/home/student'
>>> 
```

Using iPython:

In [1]: import os

In [2]: os.getcwd()

Out[2]: '/home/student'

Continues…

Using the *os* Module to Interact with Data

We will now create a folder:

```
In [3]: os.mkdir("/tmp/os_mod_explore")
```

List the contents

```
In [4]: os.listdir("/tmp/os_mod_explore")
Out[4]: []                                   #nothing, as folder is empty
```

We will now create a folder and list contents:

```
In [5]: os.mkdir("/tmp/os_mod_explore/test_dir1")
In [6]: os.listdir("/tmp/os_mod_explore")
Out[6]: ['test_dir1']
```

Perform stat() system call on the specified path

```
In [7]: os.stat("/tmp/os_mod_explore")
Out[7]: os.stat_result(st_mode=16893, st_ino=388743, st_dev=2049, st_nlink=3, st_uid=1000, st_gid=1000,
st_size=4096, st_atime=1585153221, st_mtime=1585145278, st_ctime=1585145278)
```

Continues…

Using the *os* Module to Interact with Data

Rename directory:

```
In [8]: os.rename("/tmp/os_mod_explore/test_dir1",
"/tmp/os_mod_explore/test_dir1_renamed")
```

List contents:

```
In [9]: os.listdir("/tmp/os_mod_explore")
Out[9]: ['test_dir1_renamed']
```

Remove directories:

```
In [10]: os.rmdir("/tmp/os_mod_explore/test_dir1_renamed")
In [11]: os.rmdir("/tmp/os_mod_explore/")
```

Continues…

Using the *os* Module to Interact with Data

In Summary:

- In In[2] we get the current working directory and
- then make a directory in line [3].
- *os.listdir* in line [4] lists the content of the newly created directory.
- Next, we use *os.stat* display file system status, [7].
- In line [8] the directory is renamed.
- In line [9], we verify that the directory was created and
- then we proceed to delete what we created by using *os.rmdir* method, [10] and [11].

- There are methods to do just about anything you would need to do to the data, including changing permissions and creating symbolic links. Please refer to the documentation or alternately, use *os.<TAB>* to view the available methods for the OS module.

Copying, Moving, Renaming, and Deleting

- A higher-level module called ***shutil*** deals with data on a lager scale. The ***shutil*** module has methods for
  - copying,
  - moving,
  - renaming, and
  - deleting

  data just as the OS module does, but it can perform actions on an entire data tree.

Example of using shutil:

First some setup:

```
In [1]: import os
In [2]: os.chdir("/tmp")
In [3]: os.makedirs("test/test_subdir1/test_subdir2")
```

Continues…

Copying, Moving, Renaming, and Deleting

List – long format recursively:

```
In [4]: ls -lR
```

Sample output:

```
.:
total 4
drwxrwxr-x 3 student student 4096 Mar 25 16:50 test/

./test:
total 4
drwxrwxr-x 3 student student 4096 Mar 25 16:50 test_subdir1/

./test/test_subdir1:
total 4
drwxrwxr-x 2 student student 4096 Mar 25 16:50 test_subdir2/

./test/test_subdir1/test_subdir2:
total 0
```

Copying, Moving, Renaming, and Deleting

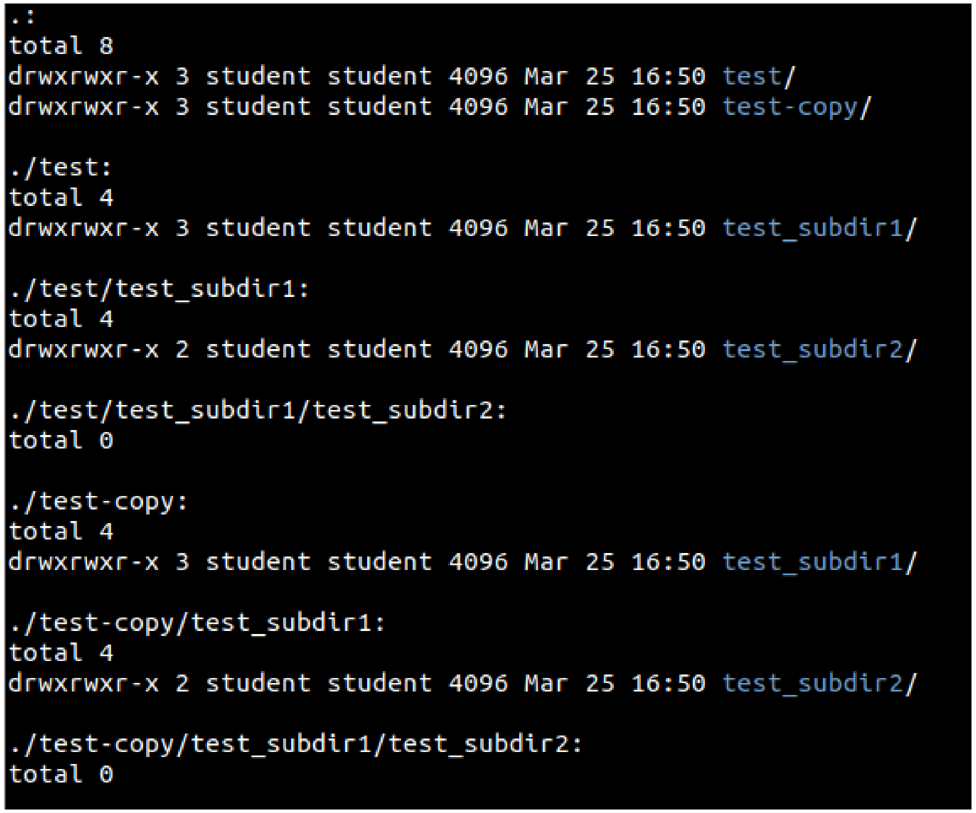
Now we import *shutil* and copy the “test” tree of directories:

In [5]: import shutil

In [6]: shutil.copytree("test", "test-copy")

In [7]: ls –lR

Sample output of ls -lR:



```
.:
total 8
drwxrwxr-x 3 student student 4096 Mar 25 16:50 test/
drwxrwxr-x 3 student student 4096 Mar 25 16:50 test-copy/

./test:
total 4
drwxrwxr-x 3 student student 4096 Mar 25 16:50 test_subdir1/

./test/test_subdir1:
total 4
drwxrwxr-x 2 student student 4096 Mar 25 16:50 test_subdir2/

./test/test_subdir1/test_subdir2:
total 0

./test-copy:
total 4
drwxrwxr-x 3 student student 4096 Mar 25 16:50 test_subdir1/

./test-copy/test_subdir1:
total 4
drwxrwxr-x 2 student student 4096 Mar 25 16:50 test_subdir2/

./test-copy/test_subdir1/test_subdir2:
total 0
```

Copying, Moving, Renaming, and Deleting

- The ***shutil*** module does not jus copy files, it also has methods for moving and deleting trees of data as well.

For example we could:

```
In [20]: shutil.move("test-copy", "test-copy-moved")
In [21]: ls –lR
```

- Example of deleting a data tree with *shutil*:

```
In [22]: shutil.rmtree("test-copy-moved")
In [23]: shutil.rmtree("test-copy")
```

Comparing Data

Comparing data, directories and files is a task a system administrator may be called upon to perform from time to time.

Questions that a **sysadmin** might often consider are:

- “What files are the differences between these two directories?
- How many copies of this same file exist on my system?”

When dealing with large volumes of important data, it often is necessary to compare not just individual files, but also entire directory trees and files to see what changes have been made.

In this section we will look at examples of this, using the *filecmp* module.

Using the *filecmp* Module

- The *filecmp* module contains functions for doing fast and efficient comparisons of files and directories.
- The *filecmp* module will perform a *os.stat* on two files and return a
  - *True* if the results of *os.stat* are the same for both files
  - or a *False* if the results are not.
- Typically, *os.stat* is used to determine whether or not two files use the same inodes on a disk and whether they are the same size.

<u>Example:</u>

- To illustrate how *filecmp* works, we need to create three files from scratch. To do this on your VM,
  1) change into the /tmp directory,
  2) make a file called *file0.txt*, and place a “0” in the file.
  3) Next, create a file called *file1.txt*, and place a “1” in that file.
  4) Finally, create a file called *file00.txt*, and place a “0” in it.

Continues…

Using the *filecmp* Module

Compare the first two files:

```
In [1]: import filecmp
In [2]: filecmp.cmp("file0.txt", "file1.txt")
Out[2]: False
```

Compare the first and the third files created:

```
In [3]: filecmp.cmp("file0.txt", "file00.txt")
Out[3]: True
```

Using the *dircmp* Module

- The *dircmp* function of the module has a number of attributes that report differences between directory trees.

Example of using *dircmp*:

Setup:

- Two subdirectories (*dirA and dirB*) were created in the */tmp* directory and the files from our previous example were copied into each directory.
- In *dirB*, we created one extra file named file11.txt, into which we type “11”

iPython3 example:

```
In [1]: import filecmp                                   #import module
In [2]: pwd                                              # check current dir
Out[2]: '/home/student‘
In [3]: cd /tmp                                          # change to our testing location
In [4]: filecmp.dircmp("dirA", "dirB").diff_files        # check for different files
Out[4]: []
In [5]: filecmp.dircmp("dirA", "dirB").same_files        #check for same files
Out[5]: ['file1.txt', 'file00.txt', 'file0.txt']         # lists the same files
```

Comparing Data

```
In [6]: filecmp.dircmp("dirA", "dirB").report()                    # report on differences
diff dirAdirB
Only in dirB: ['file11.txt']
Identical files : ['file0.txt', 'file00.txt', 'file1.txt']
```

Summary:

- You might be a bit surprised to see here that there were no matches for *diff_files* even though we created a *file11.txt* that has unique information in it.

  The reason is that *diff_files* compares only the differences between files with the same name.

- Looking at the output of *same_files*, and you will notice that it only reports back files that are identical in two directories.

- Finally, we can generate a report as shown in the last example. It has a handy output that includes a breakdown of the differences between the two directories.

This brief overview is just a small example of what the *filecmp* module can do.

Comparing Data using *os.list*

- If you want to use *os.list* for comparing file/directory data you can think of ***os.listdir*** as an *ls* command that returns a Python list of the files found.
- As Python supports many interesting ways to deal with lists, you can use ***os.listdir*** to determine differences in a directory yourself, by simply converting your list into a ***set*** and then subtracting one set from another.

First a Side-note about sets:

- A Python ***set*** is a slightly different concept from a list. A set, in Python, is just like the mathematical set. It contains an unordered collection of objects and does not hold duplicate values.
- In Python sets uses the curly braces **{}** .

Example of a set declaration:

```
In [1]: myset = {3,2,1,4,5,5}

In [2]: myset
Out[2]: {1, 2, 3, 4, 5}
```

- As you can see in [2] when we print the set to the screen the elements are rearranged in ascending order, with only unique values.
- As the set is unordered, we can not use indexing to access the original elements.

Comparing Data using *os.list*

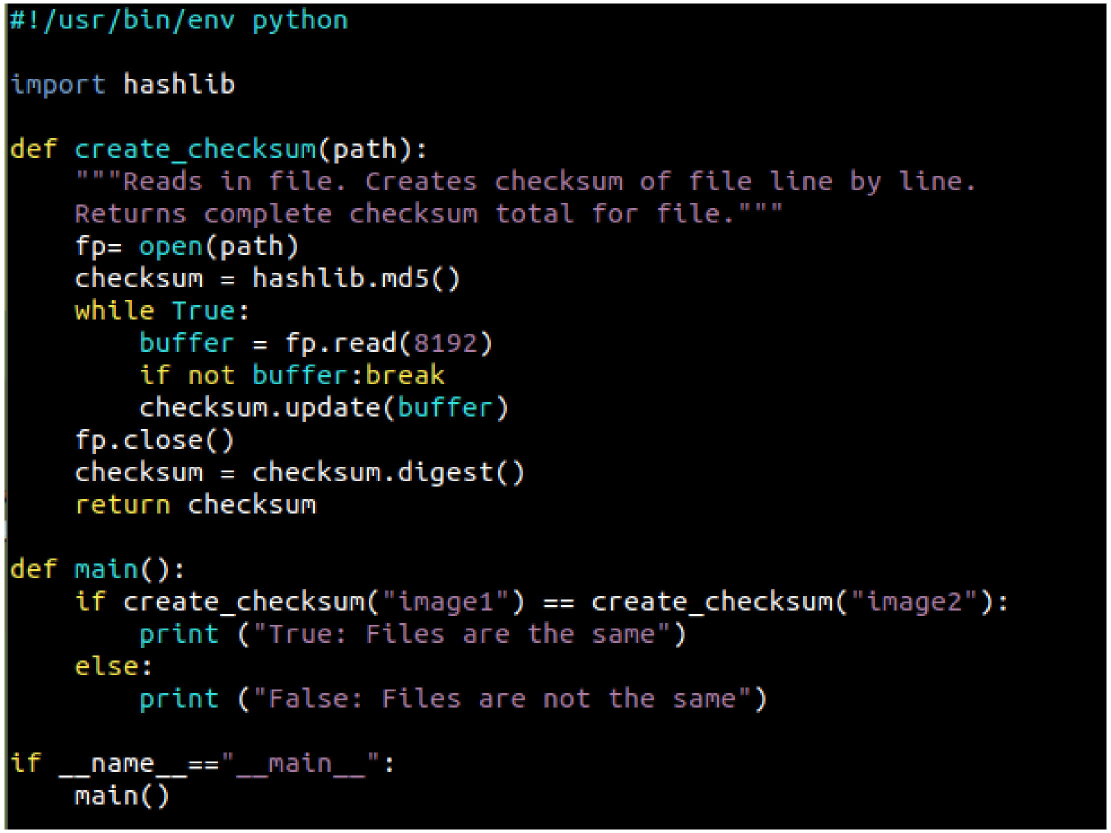
- Returning to our example of comparing directory and file data using ***os.listdir*** :

Using our previous dirA and dirB

```
In [1]: import os                                  # import os module
In [2]: dirA= set(os.listdir("/tmp/dirA"))         # set dirA to a set containing the…
In [3]: dirA                                       # …files names in /tmp/dirA
Out[3]: {'file1.txt', 'file00.txt', 'file0.txt‘}

In [4]: dirB= set(os.listdir("/tmp/dirB"))         # same for dirB
In [5]: dirB
Out[5]: {'file0.txt', 'file00.txt', 'file1.txt', 'file11.txt'}

In [6]: dirA-dirB                                  # difference between dirA & dirB
Out[6]: set()                                      # empty set

In [7]: dirB-dirA                                  # difference between dirB & dirA
Out[7]: {'file11.txt'}
```

MD5 Checksum Comparisons

- Performing a MD5 checksum on a file and comparing it to another file is a bit of an overkill for our simple example scenario. However it is very useful when you want to be sure of what you are doing, although a byte-by-byte comparison is truly 100 percent accurate.
- The following is a script "*checksum.py*" illustrates its use (*I used an image copied to files named: image1 and image2 for testing*)



```
#!/usr/bin/env python

import hashlib

def create_checksum(path):
    """Reads in file. Creates checksum of file line by line.
    Returns complete checksum total for file."""
    fp= open(path)
    checksum = hashlib.md5()
    while True:
        buffer = fp.read(8192)
        if not buffer:break
        checksum.update(buffer)
    fp.close()
    checksum = checksum.digest()
    return checksum

def main():
    if create_checksum("image1") == create_checksum("image2"):
        print ("True: Files are the same")
    else:
        print ("False: Files are not the same")

if __name__=="__main__":
    main()
```

Comparing Data

- If you want to use the *create_checksum* function from our script within iPython to do the same comparison of two files the following is an iterative example:

```
In [2]: from checksum import create_checksum
In [3]: if create_checksum("image1") == create_checksum("image2"):
...: print "True"
...:
...:
True
In [5]: if create_checksum("image1") == create_checksum("image_unique"):
print "True"
...:
...:
```

- In that example, the checksums of the files were manually compared, but we can use the code we wrote earlier that returns a list of paths to recursively compare a directory tree full of files and gives us duplicates.