IT Scripting and Automation

Python
Part 3

Lecturer: Art Ó Coileáin

Shell Execute

- Another, and possibly easier, way of executing a shell command is to place an exclamation point (!) in front of it:

```
In [5]: !netstat -lptn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
455/rpcbind
tcp        0      0 0.0.0.0:58865           0.0.0.0:*               LISTEN
464/rpc.statd
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
481/sshd
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
749/exim4
tcp6       0      0 :::111                  :::*                    LISTEN
455/rpcbind
tcp6       0      0 :::22                   :::*                    LISTEN
481/sshd
tcp6       0      0 ::1:25                  :::*                    LISTEN
749/exim4
tcp6       0      0 :::54013                :::*                    LISTEN
464/rpc.statd

In [6]: _
```

Shell Execute

- Variables can be passed to your shell commands by prefixing them with a dollar sign ($). For example:

```
In [6]: user='student'

In [7]: process='top'

In [8]: !ps aux | grep $user | grep $process
student     783  0.6  0.2   5068  2784 tty2     S+   13:02   0:00 top

In [9]: _
```

- This listed sessions of user '**student**' running **top** command.
- We can store the result of a !Command:

In [4]: l = !ps aux | grep $user | grep $process

Bookmark

- It persists across iPython sessions. If your exit iPython and start it back up, your bookmarks will still be there.

There are two ways to create bookmarks:

1. The first way is:

```
In [1]: cd /tmp
/tmp
In [2]: bookmark t
```

- By typing in bookmark **t** where you are in **/tmp**, a bookmark named **t** is created and pointing at **/tmp**.

2. The second way is:

```
In [3]: bookmark muzak /home/jmjones/local/Music
```

- It creates a bookmark named ***muzak*** that point to a local music directory.

Bookmark

- The command **bookmark -l** tells iPython to list the bookmarks.
- There are two ways of removing bookmarks. The first removes a particular bookmark and the second removes all the bookmarks.

  bookmark -d bookmark_name

  bookmark -r
- To show a list of directories visited, we use the command:

  dhist
- A simple command to tell the current directory is:

  pwd

Variable Expansion

- We have mostly kept shell stuff with shell stuff and Python stuff with Python stuff. But now, we are going to cross the line and mingle the two of them.

```
In [1]: for i in range(2):
...:         !date >> test.txt

In [2]: ls

In [3]: !cat text.txt
        Fri OCT 25 07:40:05 EST 2019
        Fri OCT 25 07:40:06 EST 2019
```

String Processing

- A powerful feature that iPython offers is the ability to string process the system shell command output. If we want to see the PIDs of all the process belonging to a particular user. In shell:

  ps aux | awk'{if ($1 == “sean") print $2}'

- In iPython, we grab the output from an unfiltered **ps aux**:

  In [1]: ps= !ps aux

- We can use the grep() method: **In [2]: ps.grep(‘sean')**

- If we want to exclude entries that match with the pattern we can use an additional argument:

  In [3]: ps.grep('Mar07', prune=True)

String Processing

- We can also use the *fields()* method, E.g.:

  In [4]: ps.grep('student').fields(0, 1, 8)

  - Here we are using on the result of the *grep()* method call. We are able to do this because *grep()* returns an object of the same type as the *ps* object that we started with and *fields()* itself returns the same object type as *grep().*

- The field() method takes a number of arguments. They are expected to be the "columns" from the output, if the output lines were split on whitespace. Similar to **awk** does to lines of text. So we call *fields()* to view columns **0**, **1** and **8**

  In [5]: ps.grep('student').fields(1).s

- The final piece of string processing is the **s** attribute of the object trying to directly access your process list. The **s** attribute gives a nice space-separated string of PIDs that we can work with in a system shell.

Possible use: We could store that string field list in a variable called pids and do something like kill $pids from within iPython.

String Processing

- An alternative option is:

```
In [1]: ps= !ps aux
In [2]: ps.grep('student', field=0)
In [3]: ps.grep('student', field=0).fields(1)
```

Creating Strings

- The most common way to create a string is to surround the text with quotation marks:

```
In [1]: string1 = 'This is a string'
In [2]: string2 = "This is another string"
In [3]: string3 = '''This is still another string'''
In [4]: string4 = """And one more string"""“
In [5]: type(string1), type(string2), type(string3), type(string4)
Out[5]: (<type 'str'>, <type 'str'>, <type 'str'>, <type 'str'>)
```

- Single, double, and triple quotation marks accomplish the same thing: they all create an object of type str. Single and double quotation marks are identical in the creation of strings; you can use them interchangeably.

- This is different from the way quotation marks work in Unix shells, in which the marks cannot be used interchangeably.

Creating Strings

- Multiple lines strings: can be created by embedding **\n** in the string or using the triple quotes can be used.
- Python call “raw” strings a string that Python does not interpret escape sequences. They are created by placing a letter **r** immediately before the quotation mark.

```
In [7]: s = '''this is a
...: multiline string'''
In [8]: s
Out[8]: 'this is a\nmultilinestring‘

In [9]: s = r'''\t'''‘
In [11]: s
Out[11]: '\\t'
```

Sequence	Interpreted as
\newline	Ignored
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII backspace
\f	ASCII form feed
\n	ASCII line feed
\N{name}	Named character in Unicode database (Unicode strings only)
\r	ASCII carriage return
\t	ASCII horizontal tab
\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\Uxxxxxxxx	Character with 32-bit hex value xxxx (Unicode only)
\v	ASCII vertical tab
\ooo	Character with octal value oo
\xhh	Character with hex value hh

Built-in methods for strdata extraction

- **in** and **not in** are used to determine if a string is a part of another string.

```
In [1]: a='Technological University Dublin'
In [2]: b='Technological'
In [3]: b in a
Out [3]: True
In [4]: b not in a
Out [4]: False
```

- If you need to know where in a string the substring occurs. Methods *find()* and *index()* are used.

```
In [5]: a.find('University')
Out [5]: 14
In [6]: a.index('University')
Out [6]: 14
```

Built-in methods for strdata extraction

In [7]: a.index('Tallaght')

ValueErrorTraceback(most recent call last)

<ipython-input-59-6817f1fb75ed> in <module>()

Value Error: substring not found

In [7]: a.find('Tallaght')

Out [7]: -1

You get -1 value if the substring was not found.

- String "*slicing*":

In [8]: a[:14]

Out [8]: 'Technological '

In [9]: a[14:]

Out [9]: 'University Dublin'

startswith() and endswith()

In [1]: some_string= “Technological University Dublin"

In [2]: some_string.startswith(“Technological")

Out[2]: True

In [3]: some_string.startswith("Thursday")

Out[3]: False

In [4]: some_string.endswith(“Dublin")

Out[4]: True

What do these lines do?

In [6]: some_string[:len(“Technological")] == “Technological "

Out[6]: True

In [7]: some_string[:len("Thursday")] == "Thursday"

Out[7]: False

In [8]: some_string[-len(“Dublin"):] == “Dublin"

Out[8]: True

In [9]: some_string[-len(“Scripting"):] == “Scripting"

Out[9]: False

strip( ) Method

Example:

- In [1]: xml_tag= "<some_tag>"
- In [2]: xml_tag.strip("<>")
- Out[2]: 'some_tag‘

- In [11]: foo_str= "<fooooooo>blah<foo>“
- In [13]: foo_str.strip("><of")
- Out[13]: 'blah‘

- This stripped "<", "f", "o", even though the characters were not in that order

The upper() and lower() methods

- They are useful when you need to compare two strings without regard to whether the characters are upper-or lowercase.

```
In [1]: mixed_case_string= "VOrpalBUnny"
In [2]: mixed_case_string== "vorpalbunny"
Out[2]: False
In [3]: mixed_case_string.lower() == "vorpalbunny"
Out[3]: True
In [4]: mixed_case_string== "VORPAL BUNNY"
Out[4]: False
In [5]: mixed_case_string.upper() == "VORPAL BUNNY"
Out[5]: True
In [6]: mixed_case_string.upper()
Out[6]: 'VORPAL BUNNY'
In [7]: mixed_case_string.lower()
Out[7]: 'vorpalbunny'
```

split() Method

- Typical use of the split() method is to pass in the string that you want to split.

Example:

```
In [1]: comma_delim_string= "pos1,pos2,pos3"
In [2]: pipe_delim_string= "pipepos1|pipepos2|pipepos3"
In [3]: comma_delim_string.split(',')
Out[3]: ['pos1', 'pos2', 'pos3']
In [4]: pipe_delim_string.split('|')
Out[4]: ['pipepos1', 'pipepos2', 'pipepos3']
```

- Multiple delimiter example

```
In [1]: multi_delim_string= "pos1XXXpos2XXXpos3"
In [2]: multi_delim_string.split("XXX")
Out[2]: ['pos1', 'pos2', 'pos3']
```

split() Method

- What if you only want to split the string on the first “n” occurrences of the specified delimiters?

Example:

```
In [1]: two_field_string= "8675309,This is a freeform, plain text, string"
In [2]: two_field_string.split(',', 1)
Out[2]: ['8675309', 'This is a freeform, plain text, string']
```

- We split on a comma and told split() to only split on the first occurrence of the delimiter

split() Method

- If you need to split on whitespace in order to retrieve, for example, words from a piece of prose-like text, split() is an easy tool for accomplishing that:

Example:

```
In [1]: prosaic_string= "Insert your clever little piece of text here."
In [2]: prosaic_string.split()
Out[2]: ['Insert', 'your', 'clever', 'little', 'piece', 'of', 'text', 'here.']
```

splitlines() Method

- *splitlines()* returns a list of each line within the string and preserved groups of "words."

Example:

```
In [1]: multiline_string= """This
...: is
...: a multiline
...: piece of
...: text"""“
In [2]: lines = multiline_string.splitlines()
In [3]: lines
Out[4]: ['This', 'is', 'a multiline', 'piece of', 'text']
```

join() Method

- If you need to piece a string together from data you already have.

Example:

```
In [1]: some_list= ['one', 'two', 'three', 'four']
In [2]: ','.join(some_list)
Out[2]: 'one,two,three,four'
In [3]: ', '.join(some_list)
Out[3]: 'one, two, three, four'
In [4]: '\t'.join(some_list)
Out[4]: 'one\ttwo\tthree\tfour'
In [5]: ''.join(some_list)
Out[5]: 'onetwothreefour'
```

replace() Method

- Replace() takes two arguments: the string that is to be replaced and the string to replace it with, respectively.

Example:

In [1]: replacable_string= "trancendentalhibernationalnation"

In [2]: replacable_string.replace("nation", "natty")

Out[2]: 'trancendentalhibernattalnatty'