Python Introduction

Python is a scripting language - much like BASH / SHELL is a list of commands.

We run Python programs using a program called `python`. There are multiple versions of python, version 3 is current standard and version I will focus on.

Python is one of many scripting languages. Others include Perl, Ruby, Rust and even the Bash/Shell programming we've been talking about.

It is a script because we write the code in a text file and then run it directly with an interpreter. Before when we ran a script in bash/shell we would do

```
bash myscript.sh
```

To run code for python we will do

```
python myscript.py
```

#A first script

Here's a text file that is called 'hello.py' and contains the following. This would be the content of a text file we called `hello.py`.

```
print("Hello World!")
```

```
$ python hello.py
hello world
$ python hello.py > message.txt  # can redirect the output
$ cat message.txt
hello world
```

The Python interpreter

The Python interpreter is a program we run, giving it a file or script which specifies the actions for it to take.

One can also just run the interpreter on the command line and interact with it directly. Just execute the python command on the command line.

```
$ python
Python 2.7.10 (default, Oct  3 2015, 13:37:56)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.72)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello world"
hello world
```

#Documentation and Comments

Comment lines start with a '#'

There are also othe self-documentation feature to make it easy to generate readable documention for web or other sites. These are flanked with three double quotes like

```
"""Documentation here"""
# can also be on multiple lines
"""This message could be concise, on multiple lines, and
later I could tell you more about
my program or function"""
```

#Python version 3

There is a "new" (released 2009) version of Python. Some syntax differences and the way that lists and some operators work is different, but very similar syntax.

The default python on HPCC is python version 2 BUT you can easily switch to default python 3 by doing this. I in fact have added these lines to my ~/.bashrc and they are the default when I log in.

```
module unload miniconda2
module load miniconda3
```

- What's different 3.0 vs 2.0 - https://docs.python.org/3.0/whatsnew/3.0.html
- What's new in 3.6? - https://docs.python.org/3/whatsnew/3.6.html

```
$ module switch python/3
$ python3
Python 3.6.0 (default, Jan 30 2017, 17:43:08)
[GCC 4.8.3 20140911 (Red Hat 4.8.3-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello world"
File "<stdin>", line 1
print "hello world"
                  ^

SyntaxError: Missing parentheses in call to 'print'
>>> print( "hello world")
hello world
```

Resources reminder

The Software Carpentry Python Tutorial Is a useful place to start.

#Let's do Some math with the interpreter

Can run python just by typing on cmdline `python`. This brings up an interactive session.

You can also use the iPython/Jupyter notebooks for web-based interactive session - see the section at end of lecture 1 on "Web Access with Jupyter".

```
>>> 3+10
13
>>> 33/13
2
>>> 33/13.0
2.5384615384615383
>>> 2**10   # 2^10
1024
>>> (909+951) / 2.0 # Parenthees make a difference
930
>>> 909+951/2.0
1384
```

Modules: for Math

Python has the concept of modules or libraries that contain routines we can re-use. We will spend more time on the types of modules that exist (the list is vast!). Using the math module which defines many routines.

```
>>> sqrt(9)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> import math
>>> math.sqrt(9)
3
>>> math.pow(2,10)
1024.0
>>> 33/13.0
2.5384615384615383
>>> math.ceil(33/13.0) # round up
3.0
>>> math.floor(33/13.0) # round down
2.0
>>> round(2.2) # round works as you expect for rounding (&lt;.5)
2.0
>>> round(2.6) # round works as you expect for rounding (&gt;=.5)
3.0
```

Other math capabilities

The % operator is modulus - it means return the remainder after dividing by the number. It is useful to see if something divides evenly into a number

```
>>> 10 / 3
3
>>> 10 % 3  # 10 / 3 is 3 with remainder of 1
1
>>> 10 / 2
5
>>> 10 % 2   # 10 / 2 is 5 with a remainder of 0
0
```

This can be useful to see if something is “Modulo” something (or Mod) – really useful when looking at sequence data and checking to see if it will translate – is the length/value mod 3 == 0 - means it is divisible perfectly by 3.

Strings in Python

Quotes can define the different special characters and use of strings

```
>>> 'break dance'
'break dance'
>>> "break dance"
'break dance'
>>> "don't break dance"
"don't break dance"
>>> 'don\'t break dance'
"don't break dance"
>>> ('concatenate ' 'these ' 'together')
'concatenate these together'
```

Variables

Data can be assigned to variables. Variables don’t exist until they are declared.

The = is used for assignment. Some modifiers allow for modifying variable in place

- += add to the value of the variable
- -=, /=, *= are other

```
>>> d = 0 # declare the variable d
>>> print("d is ", d)
d is 0
>>> print("e is ", e)
Traceback (most recent call last):
File "&lt;stdin&gt;", line 1, in &lt;module&gt;
NameError: name 'e' is not defined
```

```
>>> d=8*7
>>> print(d)
56
>>> d += 100
>>> print(d)
156
```

String functions

Documented well in the python string library code.

- string.find(substring,[,starting[,end]]) Find a substring within a string

```
>>> str = "Jumping cow over the moon"
>>> str.find("cow")
8
>>> str.find("o") # start searching from beginning
9
>>> str.find("o",12) # start searching after character 12
22
```

- `string.split(separator, max_split)`

Split a string into a list based on a separator (great for column delimited data!). max_splits specifies if it should stop after N separators are found

```
>>> str = "Jumping cow over the moon"
>>> str.split(" ")
['Jumping', 'cow', 'over', 'the', 'moon']
>>> str.split(" ",2)
['Jumping', 'cow', 'over the moon']
```

substring - extracting parts of a larger string

You can query the string with the `[ ]` operator to get a subset of the string

```
>>> msg="I am a golden god!"
>>> msg[:4] # everything before position 4
'I am'
>>> msg[7:13] # get a middle part from 7-15
'golden'
>>> msg[14:] # get from 14 to the end
'god!'
>>>msg[msg.find('am'):msg.find('!')]
'am a golden god'
```

Lists and list functions

These are useful for collecting items you can enumerate.

```
>>> l = [ 2, 3, 5]
>>> l[0] # lists start at 0
2
>>> l[1:3]
3,5
>>> l.append(10)
>>> print(l)
[2, 3, 5, 10]
```

The `sum()` function will summate a list (add all the numbers up), `len()` will report how long it is. `max()` will report the largest number in a list

```
>>> sum(l)
20
>>> len(l)
4
>>> max(l)
10
```

Sorting lists

```
>>> l = [10,3,17,4]
>>> l.sort()
>>> l
[3, 4, 10, 17]
>>> ls = ['zf','fz','no','apple']
>>> ls.sort()
>>> print(ls)
['apple', 'fz', 'no', 'zf']
```

Using split to parse strings into lists

Starting with strings containing several parts which encode information you want.

```
Symbol:NASDAQ=AAPL;Date:2015-10-01;Performance:Open=111.29,Close=109.56
Symbol:NASDAQ=MSFT;Date:2015-10-01;Performance:Open=46.65,Close=46.53
```

Extract some of the info using `split`

```
q =
"Symbol:NASDAQ=AAPL;Date:2015-10-01;Performance:Open=111.29,Close=109.56"
types = q.split(";")
print(types)
# will print
['Symbol:NASDAQ=AAPL', 'Date:2015-10-01',
'Performance:Open=111.29,Close=109.56']
symbolset = types[0].split(":")
print(symbolset)
# will print
['Symbol', 'NASDAQ=AAPL']
symbol = symbolset[1].split("=")
symbol
# will print
['NASDAQ', 'AAPL']
print(symbol[1])
# will print
'AAPL'
print(symbolset[1].split("=")[1])
# will print
'AAPL'
```

`repr` function to display an object/string/number

```
# use repr to print out literally the string
hello = 'hello world\n'
print(hello)
# would print
hello world

print(repr(hello))
# would print
'hello world\n'
```

More fancy printing

Can use formatting to print some things out like these important numbers and phrase.

A formatted string uses the `%` operator to specify placeholder. Formatted printing is also easy if you are comfortable with style used in most programming languages.

```
n = [1, 2, 'oh my god']
```

~~~
print(n)
[1, 2, 'oh my god']
# print things out with fancier printing
print("%s %s %s."%(n[0],n[1],n[2]))
# prints
1 2 oh my god.
print("%5s %5s %-10s."%(n[0],n[1],n[2]))
    1     2 oh my god .
print("%-5s %5s %-10s."%(n[0],n[1],n[2]))
# prints
1         2 oh my god .
print("%-5s %5s %20s."%(n[0],n[1],n[2]))
# prints
1         2            oh my god.


# Print out with placeholders

The format is first a __formatting__ string which contains
symbols `{}`. These `{}` are replaced, in order, by the items
that follow in the `format()` function applied.

```python
>>> a = 15
>>> b = 41
>>> print( "a + b = ", a+b)
a + b =  56
>>> print( "{} + {} = {}".format(a,b,a+b))
15 + 41 = 56
>>> print("{} ({}) + {} ({}) = {}".format("a",a,"b",b,a+b))
a (15) + b (41) = 56
~~~

Can also use `rjust` and `ljust` to right or left justify a string in place

```
>>> n = [1, 2, 'oh my god']
>>> print(n)
[1, 2, 'oh my god']
# print things out with fancier printing
>>> print(n[0].rjust(5), n[1].rjust(5), n[2].ljust(10))
AttributeError: 'int' object has no attribute 'rjust'
>>> print( repr(n[0]).rjust(5), repr(n[1]).rjust(5), n[2].ljust(10))
    1     2 oh my god
```

```
>>> print( repr(n[0]).ljust(5), repr(n[1]).rjust(5), n[2].ljust(10))
1         2 oh my god
>>> print( repr(n[0]).ljust(5), repr(n[1]).rjust(5), n[2].rjust(10))
    1     2  oh my god
>>> print( repr(n[0]).rjust(5), repr(n[1]).rjust(5), n[2].rjust(12))
    1     2    oh my god
>>> print( repr(n[0]).rjust(5), repr(n[1]).rjust(5), n[2].ljust(12))
    1     2 oh my god
```