MANGOSOLUTIONS
data analysis that delivers

Python Primer for the EARL 2015 Workshop

Dr. Chris Musselle

Consultant at Mango Solutions
cmusselle@mango-solutions.com

Contents

1 Introduction

The material in this document is intended to bring the reader up to speed on the basics of setting up and working with a Python environment, and also to provide an overview of the basic syntax and data structures of the Python programming language.

The upcoming EARL 2015 workshop is an intermediate level course, and will focus more on the use of specific Python libraries rather than the Python language itself. Therefore this material is provided to allow anyone without prior Python experience to attend with a good working knowledge of the fundamentals.

2 Installation

One of the biggest hurdles in starting out with Python is installing the core language along with all the 3rd party libraries, and getting everything configured to work together correctly. Fortunately a couple of companies now offer "Python Distributions" which install Python along with many of the extended libraries and ensure everything is configured correctly.

The distribution we will be using in this workshop is the Anaconda Python Distribution: This is the simplest way to get the Python language and many of the extended libraries setup and configured in a one click installer. It is also available cross platform and freely available.

NOTE: Prior to the workshop, participants should insure they have the Anaconda Python Distribution installed for Python 3.4

The choice of operating system is left up to the participant, though it should be noted that all workshop examples will be given for a Windows environment.

3 Typical Python Workflow

3.1 Setup

The simplest Python workflow consists of having a Python interpreter open, in which to try things out and work interactively, along with a text editor to write and build up more reproducible scripts.

In this workshop we will focus on using the Ipython QT console as the interpreter, (an enhanced version of the standard Python interpreter) which runs in a separate GUI and looks much like a terminal environment.

The Ipython QT console is packaged along with the Anaconda Python Distribution and can be started by following:

```
Start --> All Programs --> Anaconda --> Ipython QT Console
```

The font size can be adjusted using `Ctrl` + and `Ctrl` –

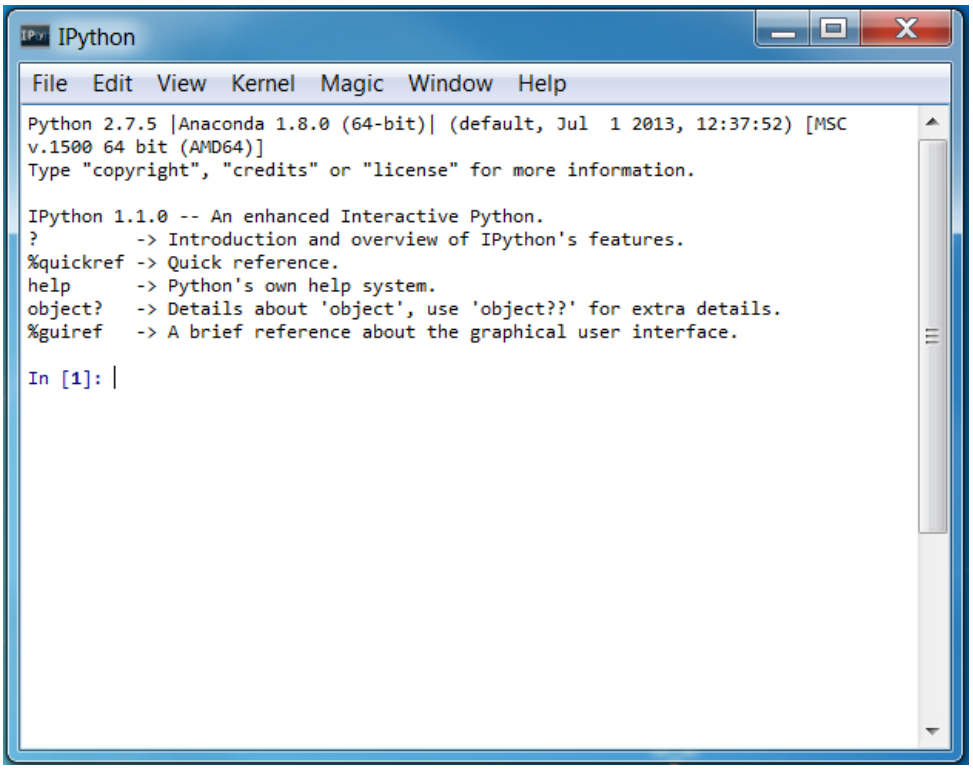

Figure 1: Fig1. - The Ipython QT console

The text editor used for the workshop is down to the participant, though having one that can automatically convert tabs to spaces is greatly advised (more on why later). Some popular free options include:

- notepad++
- Sublime (evaluation version)
- Text Wrangler (Mac OS only)

3.1.1 Spyder: A Feature Rich Python IDE

If you have time, you may wish to explore the Spyder IDE that comes with the Anaconda Distribution. This is a great IDE that will feel very familiar to any Rstudio of Matlab users. It has many helpful features, e.g. code completion, syntax highlighting, code analysis, and integrated Ipython Console.

We will not use this in the workshop for simplicity, but you are highly encouraged to explore it in your own time. It can be started by following:

```
Start --> All Programs --> Anaconda --> Spyder
```

The Spyder documentation is available `https://pythonhosted.org/spyder/`.

3.2 The Ipython Console

Ipython has many useful `Magic' functions that help perform common tasks. Some of the most helpful are detailed below, though for a full list the user can type `%quickref` (hit `q` to exit the pager.)

3.2.1 Directory Navigation

- `pwd` - - - Print working directory.
- `ls` - - - List files and directories (aliased to DIR on Windows OS)
- `cd <dirname>` - - - Change directory. (Goes to HOME if no dirname given)
- `bookmark <name>` - - - Saves an alias to the current directory so you can use `cd <name>` for quicker navigation.

3.2.2 Running a Script in the Workspace

The following will run a Python script in the Ipython Console environment. Not you will need to be in the same directory as the script itself for this to work.

```
run myscript.py
```

An alternative is to give the path to the script, though note that on Windows this will need to be given in quotes if there are any spaces in the path itself.

```
run "/path/to/my files/myscript.py"
```

3.2.3 Variable Exploration

To see information on what objects have already been defined in the namespace:

```
whos
```

To view the documentation on an IPython magic function or any python function, use `objectName?`. e.g.

```
bookmark?
```

If the amount of information spans more than one screen, it will be opened in a pager (use `Page Up/Down` or navigate and `q` or `Esc` to quit)

4 Core Python Data Types and Structures

4.1 Numbers

```
In [2]: age = 27                # Integers
        height = 1.76           # Floating point numbers
        k = 6.626e-32           # Using scientific notation
```

All standard arithmetic operations are supported.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Power
/	Division
//	Floor Division (round down after division)
%	Modulus (remainder after division)

4.2 Strings

Strings in Python are a contiguous set of characters, and can be declared with either single or double quotes.

```
In [3]: name = 'John'
        surname = "Smith"
```

For strings, the plus (+) sign will concatenate two strings into one, and the asterisk (*) will repeat a string a set number of times.

```
In [4]: print('Hello' + ' ' + 'World')
```

```
Hello World
```

```
In [5]: print('Hello' + ' ' + 'World' * 4)
```

```
Hello WorldWorldWorldWorld
```

```
In [6]: print('Hello' + (' ' + 'World') * 4)
```

```
Hello World World World World
```

4.3 Lists

The List data structure is the most versatile of Pythons built in data structures, and can hold any sequence of Python objects and mix multiple types, even other lists.

They are created using square brackets `[]`, with individual items separated by commas.

The `list()` function can also be used to convert a string to a list.

```
In [7]: mylist = [1, 2, 3, 4.0, 5, 6+2j, 'x', 'y', 'z']
        letters = list('ABCDEF')
        nestedList = [[1, 2], [3, 4], ['i', 'j']]
```

- This differs from R lists in that they are not named, and so only accessible through positional indexing.
- Lists also support the plus (+) and asterisk (*) operators for list concatenation and repetition respectively.

4.4 Indexing

- Strings and Lists are examples of sequence objects in python, which represent ordered collection of elements.
- Elements can be indexed and subset using square brackets `[]` after the variable name.
- Be aware that **indexing in Python starts from 0**.

  firstElem = variableName[0]

Negative indexing is also supported i.e. where -1 refers to the last element in the sequence and -2 the second to last element etc.

```
In [8]: msg = 'Hello World'
        msg[-5]
```

```
Out[8]: 'W'
```

4.5 Subsetting

Subsets of the elements can be returned by using *slicing notation*, where start and stop indices are specified and separated by a colon.

The elements returned from the subset include all those from the start index up to but not including the one at the stop index.

```
In [9]: msg = 'Hello World'
        msg[1:8]          # second element to the eighth
```

```
Out[9]: 'ello Wo'
```

The full slicing notation actually takes 3 parameters, the third one being the step size:

```
object[start:stop:step]
```

Thankfully, the parameters for slicing notation default to very useful values, making subsetting tasks much easier.

- If `start` is omitted, it defaults to zero.
- If `stop` is omitted it defaults to the size of the sequence being sliced.
- If `step` is omitted it defaults to one.

```
In [10]: #     Char   | H | e | l | l | o |   | w | o | r | l | d  |
         #            |   |   |   |   |   |   |   |   |   |   |    |
         #     Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
```

```
In [11]: msg[6:]        # From 7th element to the end.
```

```
Out[11]: 'World'
```

```
In [12]: msg[:5]        # First 5 elements.
```

```
Out[12]: 'Hello'
```

```
In [13]: msg[::2]       # Every other element.
```

```
Out[13]: 'HloWrd'
```

```
In [14]: msg[::-1]      # Step size can be negative, meaning the sequence is reversed.
```

```
Out[14]: 'dlroW olleH'
```

4.6 Dictionaries

A dictionary can hold any combination of Python objects as values, but instead of indexing them by position (and hence a number) as with sequence objects, values in a dictionary are unordered and indexed by keys.

It is best to think of a dictionary as an unordered set of key: value pairs, where each key must be unique, and maps to a certain value (that does not need to be unique).

Dictionaries are created by using the curly brackets notation `{}`, where each key to value mapping is specified with a colon `:`.

```
dict_name = {key1 : value1, key2 : value2 ... }
```

The values are referenced by specifying the key within square brackets

```
dict_name[key]
```

which is used to return the values themselves, and can also be used to assign new key-value pairs to the dictionary.

```
In [15]: D = {'name' : 'John',
              'age' : 32,
              'height' : 1.76,
              'workDays' : ['Mon', 'Wed', 'Fri']}

In [16]: D['age']

Out[16]: 32

In [17]: D['name']

Out[17]: 'John'

In [18]: D['workDays'][0]    # Indexing list within dict

Out[18]: 'Mon'

In [19]: D['tel'] = '01123-445566'    # Adding new values
```

4.7 Summary of Pythons Compound Data Structures

So to summarise, there are 3 main types of compound data structures that we have looked at in Python

Some of these represent sequences, which preserve the order of the items they contain, while others do not.

There are also those whose elements are directly changeable (mutable) once created, and those that are not changeable (immutable), for which a new object must be created that contains the required changes.

Data Structure	Description	Mutable	Ordered
String	An ordered sequence of characters	No	Yes
List	An ordered collection of elements	Yes	Yes
Dictionary	An unordered collection of keys-value pairs.	Yes	No

5 Control Flow in Python

> Indentation is *enforced* in Python, as it is used to delimit code blocks.
>
> There is no `end` keyword or curly braces in Python.

The amount of indentation does not matter, so long as it is the same for all lines in the code block. Typically 4 spaces are used. Python IDEs and most text editors will have the ability convert tabs to spaces for convenience. For safety though, be consistent, and never mix tabs and spaces.

Failure to indent properly will result in an Error.

5.1 if statements

```
In [20]: x = 5

         if x > 0:
             print('x is positive')
         elif x < 0:    # Optional and multiple elif branches possible
             print('x is negative')
         else:          # Also optional
             print('x is zero')
```

```
x is positive
```

5.2 Boolean Operators

Operator	Description
`==`	Checks for equality.
`!=`	Checks for non-equality.
`>`	Checks if the left value is greater than the right.
`<`	Checks if the left value is less than the right.
`>=`	Checks if the left value is greater than or equal to the right.
`<=`	Checks if the left value is less than or equal to the right.
`and`	Logical AND operator
`or`	Logical OR operator
`not`	Logical negation operator. Reverses the logical state of its operand.
`in`	Membership test for sequences and dictionaries.

5.3 Python ``Truthiness''

Most python objects have a notion of ``truth'' associated with them, where the object itself is evaluated as either true or false depending on the objects contents.

The following will evaluate to `False` in an if statement: * None * 0 * `''` , `[]`, or `{}` (an empty string, list or dictionary)

Conversely, any integer greater than zero, and any non empty object will evaluate to `True`

5.4 Iterables and For Loops

An object is said to be iterable in Python if it has a method to return its individual elements in a specific order.

All sequences are iterable (Strings and Lists), as well as file objects (more on this later.)

For each iteration of the for loop, the variable `varname` is assigned to the next element in the iterable object.

```
for varname in iterable:
    statements
```

The `range` function is commonly used to create a list of values to iterate over with a similar syntax to slicing notation.

```
range(stop)      # list containing integers 0 to 'stop' - 1
range(start, stop, step)  #  list containing integers 'start' to 'stop' -1
                          #  in increments of 'step'
```

```
In [21]: # Countdown Example
         for idx in range(10,0,-1):
             print(idx)
             if idx == 1:
                 print('Lift Off!!!')
```

```
10
9
8
7
6
5
4
3
2
1
Lift Off!!!
```

6 Object Attributes and Methods

Everything in Python is an object, and all objects can have attributes which are simply variables attached to the object. If the attribute is a function it is also called an object method.

Attributes are accessed via the dot . operator.

We have already encountered several object methods in Python for file objects, and many exist for stings, lists and dictionaries as well.

In IPython, *Tab Completion* can be used after the dot . operator to list the methods and attributes of an object. Tab completion also works for variable names already defined.

6.1 Some Useful String Methods

- `split` takes a string and returns a list of the split it up by the given character(s).

```
In [22]: s = '192.168.0.1'
         s.split('.')
```

```
Out[22]: ['192', '168', '0', '1']
```

- `join` takes a list of strings and returns the joined string delimited by the object string.

```
In [23]: '---'.join(['A', 'B', 'C'])
```

```
Out[23]: 'A---B---C'
```

- `strip` removes a set of characters from the beginning and end of the string.
- See also `rstrip` and `lstrip`.

```
In [24]: s = '<My string! And some random punctuation!!?!>'
         s.strip('<>!?')
```

```
Out[24]: 'My string! And some random punctuation'
```

- `startswith` performs efficient sting matching to start of the string.
- `endswith` performs efficient sting matching to end of the string.

```
In [25]: s = 'myfile.txt'
         if s.endswith('.txt'):
             print('File is a text file')
```

```
File is a text file
```

- `replace` returns a new string with a substring replaced by a given new string.

```
In [26]: s = 'A---B---C'
         s.replace('-', '')
```

```
Out[26]: 'ABC'
```

- `find` returns the lowest index to a substring if it is present. Returns -1 otherwise.

```
In [27]: s = 'Mississippi'
         s.find('si')
```

```
Out[27]: 3
```

- Can also use the `in` operator to check for substring membership.

```
In [28]: if 'red' in 'Fred':
             print('Fred is red?')
```

```
Fred is red?
```

7 File I/O

A handle to a file object is created using the `open` function:

```
f = open(name, mode)
```

The `mode` can be `r` for reading (default), `w` for writing, or `a` for appending. A `b` is used to specify a binary file e.g. `rb`. Files are created for `w` or `a` if it does not exist.

7.1 File Object Methods

The following functions are attached to the file object created, and accessed using the dot `.` operator.

`f.read(size)` Reads at most `size` bytes from the file and returns them as a string. If not specified, the entire file is read in as one long string.

`f.readlines()` Returns a list of stings, one for each line in the file.

`f.write(string)` Writes the string to the file. Note, this does not add a newline character (`\n`) to the end of the string.

`f.writelines(sequence_of_strings)` Writes a list of string to the file. Again, this method does not add a newline character (`\n`) to the end of each line automatically.

`f.close()` flushes any remaining data to the file and prevents further writes. Python closes files automatically when their variable name is reassigned or it goes out of scope.

```
In [29]: # Example of writing and reading a text file.
         temp = open('temp.txt', 'wb')
         temp.write('Hello\nWorld\nFrom\nPython.')
         temp.close()

         # Open Text File
         f = open('temp.txt')

         for line in f:
             # Do something with each line of the file
             print(line)
```

```
Hello

World

From

Python.
```

8 Example: Using File I/O and String Methods

Given some input text, say we wished to write this file, and then a t a later date, read it back in and filter it in some way. The steps involved are:

- Split the text into words and write each word to a new line in a file.
- Read in the words from the file, and print only those with > 3 characters.

```
In [29]: # Can use parentheses to break up a string into separate lines
         # for readability. String within are concatenated.
         sample_text = ("'and what is the use of a book,' thought Alice 'without "
                        "pictures or conversations?'")

         # Split function will split the string by white space characters (by default)
         word_list = sample_text.split()

         # Open a file and write each word to a new line in the file
         output_file = open('alice.txt', 'wb')

         for word in word_list:
             output_file.write(word + '\n')

         # Close file
         output_file.close()
```

```
In [31]: # Re-open file and read in the contents to a list
         input_file = open('alice.txt', 'rb') # Note reading mode

         # Loop over input file
         for line in input_file:
             # Print only words that contain greater than three characters
             x = line.strip(",'.\n?")  # Remove starting and trailing punctuation.
             if len(x) > 3:
                 print(x)

         input_file.close()
```

```
what
book
thought
Alice
without
pictures
conversations
```

9 Imports and Modules

A module in Python is another name for a library, and is simply a .py file that contains code to define additional functions or classes. We will not look at creating functions or classes today, but we can use existing code by importing these modules into our Namespace.

This is done by using the import statement in one of three ways.

1. `import modualName`
2. `import modualName as alias`
3. `from modualName import object1, object2, object3`

The method used to import will effect how the objects imported are used. Some examples are shown bellow.

```
In [32]: import math

         math.sin(math.pi/3)

Out[32]: 0.8660254037844386

In [33]: import math as m

         m.sin(m.pi/3)

Out[33]: 0.8660254037844386

In [34]: from math import sin, pi

         sin(pi/3)

Out[34]: 0.8660254037844386
```

10 Getting Help

Help doc string for functions and object methods is available interactively in the interpreter by appending a `?` to the function name. E.g.

```
myStringVariable.join?
```

The documentation is opened in a pager if there is a lot of text. This can be navigated via the arrow keys, spacebar or page up/down. Press `q` to quite the pager and go back to the interpreter.

10.1 HTML Docs

Extended HTML versions of module documentation are also available. Some common 3rd party modules are linked to below.

- core python
- numpy - Numerical computing library.
- pandas - Manipulation of tabular data structures (data frames and series).
- matplotlib - The most common plotting library in Python.
- Ipython - An enhanced version of the standard python interpreter.