Python_Introduction_Part_1

October 9, 2019

1 Introduction to Python

This is an introduction to Python! This is the first in a set of Jupyter notebooks that will provide you with an brief overview of Python. The range of things that are possible with Python is huge, so it is not possible to cover all of them in a few lectures (or even a year long module!!). Hopefully these notebooks provide you with enough information to start your journey into scientific data analysis.

As a companion to these notes, I suggest the following books:

Python Data Science Handbook - Jake VanderPlas (also available online)

Python for Data Analysis - Wes McKinney

During the notes I also provide links to many online resource that are useful learning Python. I recommend exploring these websites in more detail to help with your learning.

If you find any useful resources that help you understand an aspect of Python better, you can help contribute to these notes. Please let me know what you've found and the future years students will benefit from your contribution.

2 Launching the Jupyter Environment

The lecture notes for this course are given in a Jupyter notebook. Jupyter is a useful tool for helping you to build functioning programs and share code with colleagues.

The lecture notes are also interactive - you can change the code and see how it behaves. I suggest you make a copy of the notebooks if you want to play around with the code.

The lecture notes can be downloaded from the Blackboard page for this module. They are all contained in zip folder called *Python_lecture_notes.zip*.

You can launch the Jupyter environment by searching for the Jupyter in the task bar. This will open up a window in you browser showing the following:

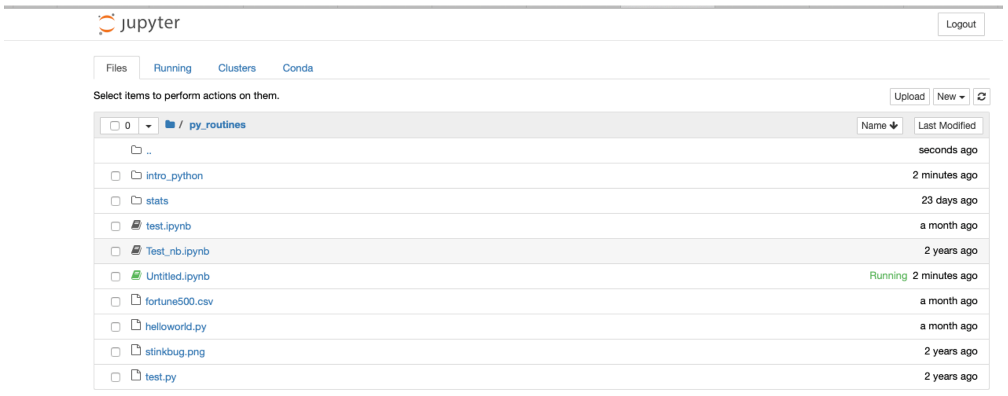
You now have to navigate to the downloaded folder and click on Python_intro_week_1.ipynb

There are many useful resources online that will help you get to grips with using Jupyter.

---

3 Launching Pycharm and Python

You can work with Python in many ways. We will use Pycharm - which is what is known as a *developer environment* and contains many useful features.



Select items to perform actions on them.

/ py_routines

..		seconds ago
intro_python		2 minutes ago
stats		23 days ago
test.ipynb		a month ago
Test_nb.ipynb		2 years ago
Untitled.ipynb	Running	2 minutes ago
fortune500.csv		a month ago
helloworld.py		a month ago
stinkbug.png		2 years ago
test.py		2 years ago

screenshot

You will have a Pycharm launch icon on your desktop. When you run this for the first time you be asked to create a project and name it.

Call it something sensible, like *KL7002_astromodule_intro* .

The Pycharm environment should open and you will presented with a Terminal console, Python console and an editor window, amongst other things.

We'll first focus on the terminal. If we click on the terminal tab, it tells us we are working in the directory username/PycharmProjects/astromodule. The last part refers to the current project directory.

3.1 IPython

We will want to work with a version of Python called the IPython shell - which basically makes the basic version of python 'pretty'.

In the terminal window, enter the following in to the command line

```
>ipython
```

We are now in a Python shell and we are able to execute python statements and commands.

E.g. input a=6 into the command line, enter, then a followed by enter. It should look like the following

```
>In[1]:a=6

>In[2]:a

>Out[2]:6
```

You can exit the IPython environment with:

```
>exit()
```

3.2 Python files

In Project tab, right click on the project root (in my examples called *astro_mod*), select *new* then *Python file*.

This will then open a new tab and ask you to name it. Let's start off with the classic *hello_world* (you will notice the files automatically got the extension *.py*, to show it is a python file).

In the file enter the following:

```
print('Hello World!')
```

and save (*ctrl+s*).

Now we return to the Python shell and you can run your newly created Python file with the %run command

```
> %run hello_world
```

---

4 Data types

In order to undertake any form of analysis you will need to work with some form of data! In its most basic variety, this will usually take the form of numbers or letters.

In Python, we call fundamental bits of data *values* and they can have different classes. For example, the number 4 belongs to the class *integer* and 'Hello' belongs to the class *string*.

We can find out the class of the values, we can use a *function* called type

```
In [1]: type('Hello')
```

```
Out[1]: str
```

Notice that the string has single quotation marks surrounding it. Double quotation marks can also be used ("hello"). If required, you can also use three of each, e.g. """Hello"""

A single number is an integer:

```
In [2]: type(4)
```

```
Out[2]: int
```

You can also have floats, which represent decimals:

```
In [3]: type(5.67)
```

```
Out[3]: float
```

If inputing a large integer, you do not need commas, e.g., 97,000 should be input as 97000. You can see what happens if the comma is included!

```
In [4]: 42000
```

```
Out[4]: 42000
```

You can also have complex numbers:

```
In [5]: type(1+2j)
```

```
Out[5]: complex
```

---

5 Variables

A variable is a way to associate a name with a value and is performed using the = sign (this is referred to as an assignment token and not to be confused with equals, which is ==).

In the following I assign the word 'cat' to variable *a* and the float 4.5 to variable *b* . I am also using the *print* function, which prints variables (and other things) to screen.

```
In [6]: a='cat'
        print(a)
        b=4.8
        print(b)
```

```
cat
4.8
```

Variables, as the name suggests, can have there value changed.

```
In [7]: month='January'
        print(month)
        month="February"
        print(month)
```

```
January
February
```

Variable names can be of any length and can contain both letters and numbers (although they have begin with a letter). They are also case-sensitive, i.e., *month* is different from *Month*. The convention is to use lower-case letters only though. The underscore can also be used in variables names, e.g., month_1985.

Improperly defined variables result in a syntax error. For example, if your variable starts with a number:

```
In [8]: 1month='December'
```

```
          File "<ipython-input-8-6613a1c621f7>", line 1
        1month='December'
             ^
    SyntaxError: invalid syntax
```

You may also get a syntax error if you choose a variable name that is a **Python keyword**.

Finally, when you define variables in a code, the name should be useful for a human to read! If you just give it a random name or *var1*, *var2*, *var3*, etc., your code will be very difficult to follow.

Be kind to your future self - make your code easy to read!

"Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write."

Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

---

6 Expressions

An expression is a combination of variables, values, operations or functions.

We will begin by looking at expressions containing mathematical operators.

6.1 Mathematical Operators

Addition

```
In [9]: 2+3
```

```
Out[9]: 5
```

Subtraction

```
In [10]: 9-10
```

```
Out[10]: -1
```

Multiplication

```
In [11]: 2*3
```

```
Out[11]: 6
```

Division

```
In [12]: 10/2
```

```
Out[12]: 5.0
```

Raising exponents

```
In [13]: 3**3
```

```
Out[13]: 27
```

Finding the remainder after division using the modulo operator

```
In [14]: 18%4
```

```
Out[14]: 2
```

We can perform the same operations with variables that are numerical

```
In [15]: a=9
         b=10
         c=a*b
         print(c)
```

```
90
```

There is also something called *floor division*, // , which performs integer division:

```
In [16]: print(7/2)
         print(7//2)
```

```
3.5
3
```

6.2 Boolean values, logical and relational operators

A Boolean value is either True or False. They are usually the outcome of testing statements with relational operators.

Here are the most common relational operators, and their Boolean output

```
In [17]: print(5==3) # This says is 5 equal to 3
         print(5!=3) #           is 5 not equal to 3
         print(5<3)  #           is 5 less than 3
         print(5>3)  #           is 5 greater than 3
         print(5<=3) #           is 5 less than or equal to 3
         print(5>=3) #           is 5 greater than or equal to 3
```

```
False
True
False
True
False
True
```

The Boolean values are useful for sorting data and conditional execution of statements - all of which we will see later.

We can combine the relational expression using logical operators, AND, OR and NOT

```
In [18]: print(5>2 and 5<7)
         print(5>2 and 5>7)
         print(5>2 or 5<7)
         print(not(5>2 and 5<7)) #Reverses the boolean, i.e. if true gives false
```

```
True
False
True
False
```

6.3 Operations with strings

You can also manipulate strings in Python.

```
In [19]: word='Alphabet'
         word2='Soup'
         print(word+' '+word2)
         print(word*3)
```

```
Alphabet Soup
AlphabetAlphabetAlphabet
```

This includes accessing specific positions in the string (counting starts at 0):

```
In [20]: print(word[3])
```

```
h
```

There are many more things that can be achieved with string manipulation, however this will not be a focus of this course. But for now it's enough you know it exists!

---

7 Storing data

You will often be working with many bits of data that come in various forms. Python contains a number of ′containers′ that will hold data (these are known as compound data types).

7.1 Lists

The most versatile, but maybe not the most appropriate for data analysis, is the *list*. The list is defined by comma separated entries enclosed in square brackets. It **can contain a mix of data types** - although it typically should only be used to contain one type.

```
In [21]: lis=[5,'apple',6.7,'goat']
```

Elements in the list are accessed by a numerical value, starting at zero. Negative indices count backwards.

```
In [22]: print(lis[0],lis[3])
         print(type(lis[0]),type(lis[3]))
         print(lis[-1])
```

```
5 goat
<class 'int'> <class 'str'>
goat
```

Lists can be *sliced* to create new lists. The colon is used to do the *'slicing'*:

```
In [23]: lis=[0,1,2,3,4,5,6,7,8,9]
         print(lis[0:4])
         print(lis[-3:])
```

```
[0, 1, 2, 3]
[7, 8, 9]
```

You can also create lists of lists. Suppose we have the following information on peoples heights (in m). Jeff - 1.60, Paul - 1.80, Sally - 1.65 . We can store this as a list of lists, e.g.

```
In [24]: heights=[['Jeff',1.60],
          ['Paul',1.80],
          ['Sally',1.65]
         ]
         print(heights[0])    # Give backs Jeff's details
         print(heights[1][1]) # Gives back Paul's height
```

```
['Jeff', 1.6]
1.8
```

7.2 Tuples

There is a special type of list called a *tuple* - which is immutable (see Section 7.5). Basically you cannot change any aspects of a tuple, e.g. size, content. They should be used if you want to avoid data in them being changed.

They are defined with parenthesis rather than square brackets.

```
In [25]: atup=('the','number','is',10)
         print(atup[0])
         atup[3]=13              # trying to change an element value results in error
```

```
the



        ---------------------------------------------------------------------------


        TypeError                                 Traceback (most recent call last)


        <ipython-input-25-93385fea8580> in <module>()
```

```
      1 atup=('the','number','is',10)
      2 print(atup[0])
----> 3 atup[3]=13              # trying to change an element value results in error


    TypeError: 'tuple' object does not support item assignment
```

As with lists, tuples can store a mixture of data types:

```
In [26]: mytup=tuple(([1,2],'text',False))
```

You can ’unpack’ a tuple, which is a useful operation when dealing with many functions in Python. To unpack a tuple you do the following:

```
In [27]: alist,sometext,abool=mytup
         print(alist)
         print(sometext)
```

```
[1, 2]
text
```

7.3 Dictionaries

A dictionary is a collection of data types that are unordered and indexed (actually, in Python versions greater than v3.5, dictionaries are now ordered!). They are defined with curly brackets and have **key** and **value** pairs.

```
In [28]: mydict={
             "fruit": "Apple",
             "value": 1.20,
             "quantity": 100,
         }
         print(mydict)
```

```
{'value': 1.2, 'fruit': 'Apple', 'quantity': 100}
```

Here the keys are "value", "fruit" and "quantity". The values are "Apple", 1.20, and 100.
You use the keys to access the values.

```
In [29]: print(mydict['fruit'])
```

```
Apple
```

If we want to find all the keys of values in the dictionary, we can use some in-built methods:

```
In [30]: print(mydict.keys())
         print(mydict.values())
```

```
dict_keys(['value', 'fruit', 'quantity'])
dict_values([1.2, 'Apple', 100])
```

We can explore the dictionary in a number of ways. One example is using a *for* loops to iterate through the collection of keys.

```
In [31]: for key in mydict:
             print(key)
```

```
value
fruit
quantity
```

Here is another example looking at the values or type for each key:

```
In [32]: for key in mydict:
             print(mydict[key],type(mydict[key]))
```

```
1.2 <class 'float'>
Apple <class 'str'>
100 <class 'int'>
```

We can check if a key exists in the dictionary

```
In [33]: if 'fruit' in mydict:
             print("It's here")     #Here I have used double quotation marks for the string
                                    #as the text contains a '
```

```
It's here
```

To add a new key and value we can do the following

```
In [34]: mydict['color']='red'
         print(mydict)
```

```
{'value': 1.2, 'color': 'red', 'fruit': 'Apple', 'quantity': 100}
```

For more on dictionaries, see e.g. *https://realpython.com/iterate-through-dictionary-python/*

7.4 Arrays

The most common way of working with numerical data is to use an array. Arrays are not native to Python but can be created and manipulated with add-on libraries. The main library for this is *numpy*.

First, let us import the *numpy* library. Full details on the *numpy* library can be found at https://docs.scipy.org/doc/numpy/index.html

```
In [35]: import numpy as np # The second part of the this statement lets us refere to a
                            # library with a shorthand notation, e.g, np
```

7.4.1 Array creation

We can create an array in many ways (depending on what you want from the array). A simple 1D array is given by providing a list to the numpy *array* function:

```
In [36]: arr1=np.array([6,7,8]) # provide a list of numbers to the np.array function
         print(arr1)
         print(type(arr1))
```

```
[6 7 8]
<class 'numpy.ndarray'>
```

A 2D is given by a list of tuples, where each tuple is a row. Hence the following matrix

$$\begin{pmatrix} 6 & 7 & 8 \\ 1 & 2 & 3 \end{pmatrix}$$

is given by

```
In [37]: arr2=np.array([(6.,7.,8.),(1.,2.,3.)])
         print('Shape of array is',arr2.shape)       # what is the size of the array
         print('Dimensions of array are',arr2.ndim)  # how many dimensions does it have
         print('Size of array is ',arr2.size)        # how many elements does it have
         print('Sum of columns is ',arr2.sum(axis=1)) # sum the elements for each row
                                                      # (axis=1 operates over columns)
```

```
Shape of array is (2, 3)
Dimensions of array are 2
Size of array is  6
Sum of columns is  [21.  6.]
```

The last four commands are known as Section **??** of the array and tell us some information about the array.

We are not limited to real numbers:

```
In [38]: c = np.array( [ [1,2], [3,4] ], dtype=complex )
         c

Out[38]: array([[1.+0.j, 2.+0.j],
                [3.+0.j, 4.+0.j]])
```

We can also create arrays of zeros or ones easily with *numpy*:

```
In [39]: np.zeros((2,3))

Out[39]: array([[0., 0., 0.],
                [0., 0., 0.]])

In [40]: np.ones((2,3))
```

```
Out[40]: array([[1., 1., 1.],
                [1., 1., 1.]])
```

We can also create arrays containing ordered values:

```
In [41]: np.arange(0,10,1)

Out[41]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [42]: b=np.arange(0,1,0.1).reshape((2,5)) # There are other recommended ways to create
                                             # floating point arrays in numpy
         print(b)
         print('Shape of b is',b.shape)

[[0.  0.1 0.2 0.3 0.4]
 [0.5 0.6 0.7 0.8 0.9]]
Shape of b is (2, 5)
```

7.4.2 Basic operations

Basic arithmetic operations can be performed on arrays.

```
In [43]: a=np.array([5,10,15,20])
         b=a+4
         print(b)

[ 9 14 19 24]
```

Multiplication is performed element wise, as opposed to typical matrix multiplication. However there are ways to perform linear algebra, e.g. the dot product, but we will not discuss them here.

```
In [44]: c=a*b
         print(c)

[ 45 140 285 480]

In [45]: print(3*np.sin(a)) # here we use numpy to get access to mathematical
                            # functions, e.g. sin

[-2.87677282 -1.63206333  1.95086352  2.73883575]
```

There are more basic mathematical functions available with *numpy*: https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.math.html

We can access the different elements of the array by indexing, one-dimensional arrays are similar to lists. There are also tricks to slice arrays and manipulate them (https://docs.scipy.org/doc/numpy/user/quickstart.html).

The following type of actions are called slicing.

```
In [46]: a=np.arange(10)
         print(a[0:5:2]) # this selects every 2nd element from 0 to 5
         print(a[::-1])  # this reverses the array

[0 2 4]
[9 8 7 6 5 4 3 2 1 0]
```

Notice what happens when we use the indicies 0 to 5 to select array elements:

```
In [47]: print(a[0:5])

[0 1 2 3 4]
```

```
In [48]: print(a[:-1])
         print(a[:])

[0 1 2 3 4 5 6 7 8]
[0 1 2 3 4 5 6 7 8 9]
```

We can also use Boolean logic to help us investigate arrays

```
In [49]: print(a<5)      # This is what happens when we use a comparison operator with
                         # the array
         print(a[a<5])   # Now we use logic to return the values that satisfy our
                         # comparison (this is not slicing)
         print(a[a==5])
         print(a[a>5])

[ True  True  True  True  True False False False False False]
[0 1 2 3 4]
[5]
[6 7 8 9]
```

What happens if we want to know the array locations where a statement is true? (Note - the following is an equivalent to IDL *where* function - the comma after the index unpacks the tuple).

```
In [50]: index,=np.where(a<5)
         print(index)
         index2,=np.where( (a>2) & (a<5) )
         print(index2.size,index2)

[0 1 2 3 4]
2 [3 4]
```

For multi-dimensional arrays, the indexing and slicing is similar but requires extra terms. (Note: array referencing is opposite to IDL!)

```
In [57]: a=np.arange(0,10,1)
         a.resize(3,3)       # Create 2D array of size 3 by 3
         print(a)
         print('Elements in first row',a[0,:])     # Select all elements in first row
         print('Elements in second column',a[:,1]) # Select all elements in second column
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Elements in first row [0 1 2]
Elements in second column [1 4 7]
```

```
In [58]: a.sum(axis=1)
```

```
Out[58]: array([ 3, 12, 21])
```

7.4.3 Array concatenation

Different arrays can be combined together via concatenation.

I demonstrate two options here. We will investigate the attributes of new array from the concatenation.

```
In [59]: a=np.floor(10.*np.random.random((2,2))) # Create a 2 by 2 array of random numbers
                                                 # and use floor operator to round values down
         print(a)
         print('Shape',a.shape)
         b=np.floor(10.*np.random.random((2,2)))
         print(b)
         c=np.hstack((a,b))      # horizontal stacking (coulmn wise)
         print('New array \n',c) # \n indicates start a new line
         print('Shape',c.shape)
```

```
[[9. 3.]
 [0. 6.]]
Shape (2, 2)
[[7. 5.]
 [3. 2.]]
New array
 [[9. 3. 7. 5.]
 [0. 6. 3. 2.]]
Shape (2, 4)
```

```
In [60]: c=np.vstack((a,b))  # Vertical stack (row wise)
         print('New array \n',c)
         print('Shape',c.shape)
```

```
New array
 [[9. 3.]
 [0. 6.]
 [7. 5.]
 [3. 2.]]
Shape (4, 2)
```

A thing to note: when operating and manipulating arrays (this also applies to lists), some actions copy the data into a new, unique array (operations of new array don't influence previous version). However, some actions do not create a unique copy of the array (operations on the new array affect the old array).

```
In [61]: a=np.floor(10.*np.random.random((2,4)))
         print('Shape of a is ',a.shape)
         b=a                   # no new object is created

         print(id(a),id(b))  # Python gives each object a unique identfier,
                             # hence if a and b are the same, the id should be the same

         print('Is b the same object as a? ',b is a)  # is checks whether two things
                                                      # are the same object
         b.shape=4,2
         print('New shape of a is ',a.shape)      # Changing shape of b changes a
```

```
Shape of a is  (2, 4)
4592075360 4592075360
Is b the same object as a?  True
New shape of a is  (4, 2)
```

Similar behavior happens with new arrays created by slicing (called *shallow* or *view* copies).

```
In [62]: a=np.floor(10.*np.random.random((2,4)))
         print('This is a\n',a)
         s=a[0,:]
         print('This is s\n',s)
         s[0]=20.
         print('This is a\n',a)      # a's data changes
```

```
This is a
 [[4. 7. 9. 2.]
 [9. 8. 0. 1.]]
This is s
 [4. 7. 9. 2.]
This is a
 [[20.  7.  9.  2.]
 [ 9.  8.  0.  1.]]
```

To make a copy, you need to use the copy method

```
In [63]: a=np.floor(10.*np.random.random((2,4)))
         b=a.copy()
         print('Is b the same object as a? ',b is a)
         print(b)
         b.shape=(4,2)    # Change shape of b
         b[0,0]=20.       # Change value in b
         print('This is a\n',a)
         print(a.shape)
```

```
Is b the same object as a?  False
[[4. 1. 5. 3.]
 [1. 9. 0. 3.]]
This is a
 [[4. 1. 5. 3.]
 [1. 9. 0. 3.]]
(2, 4)
```

7.5 Mutable vs immutable

Different data types are either mutable (can change) or immutable (can't be changed). For example a list is mutable and a string isn't mutable.

The following two examples show this. Trying to change a string creates an error:

```
In [64]: string='Hello'
         string[1]='a'  # Trying to change second letter of Hello to a
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-64-071dcd339b16> in <module>()
          1 string='Hello'
    ----> 2 string[1]='a'  # Trying to change second letter of Hello to a


        TypeError: 'str' object does not support item assignment
```

```
In [65]: list_example=[1,2,3]
         list_example[1]='grape'
         print(list_example)
```

```
[1, 'grape', 3]
```

7.6 Objects, Attributes and Methods

While we will not go deeply into this aspect, I note that Python is an object-orientated language. Hence, that is why we said we have, e.g. an integer class (Section 4) - where a class refers to a group of objects with common properties.

Given that Python is an object-orientated language, most data types and containers have two features known as attributes and methods.

Attributes are properties of the class.

Methods are essentially operations or functions that belong to the objects data.

We have already encountered attributes and methods before. You can see some of the attributes and methods of each object by using tab-complete. The available attributes and methods differ for different objects.

```
In [66]: list_example.
```

```
          File "<ipython-input-66-2568b5f369c2>", line 1
        list_example.
                     ^
    SyntaxError: invalid syntax
```

```
In [67]: class Point:
             """ Point class represents and manipulates x,y coords. """

             def __init__(self, x=0, y=0):
                 """ Create a new point at x, y """
                 self.x = x
                 self.y = y

             def distance_from_origin(self):
                 """ Compute my distance from the origin """
                 return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

```
In [68]: p=Point(3,4)    # create an object p
         print(type(p))
         print(p.x,p.y) # attributes
         print(p.distance_from_origin()) # method
```

```
<class '__main__.Point'>
3 4
5.0
```