

## Practical Session 1

### 1. Aims of the Session

In this session, you will learn how to use the **Jupyter** notebook and how to manipulate data in Python using **Numpy**. At the end of this session, you should know 1) how to open the **Jupyter** notebook and how to run Python code from it; 2) how to generate random arrays and access elements in array using **Numpy**; 3) how to do matrix operations using **Numpy**; 4) how to plot data using **Matplotlib**.

### 2. Starting, Using and Leaving Python

In this module, you will use the syntax of Python 3. There are many options for development environments for Python. You will use the Jupyter notebook, which is a browser-based graphical interface to the IPython (*Interactive Python*) shell.


- Click the **window icon** on the bottom left corner:
- Search **Anaconda Navigator** by typing **Anaconda** from the search box under All Programs.
- Click on **Anaconda 3.6 Navigator**
- Click **Launch** under Jupyter from Anaconda Navigator
- Once the jupyter notebook is open, what you can see are files and/or folders under C:\Docs
- Click **New** from Jupyter, then select **Python 3** from the pop-up list. You should see a new page with a blank cell, where you can type in Python code.
- Click File and select Rename from the pop-up list, you may give the file a name you prefer, for example, practical1.

### 3. Help within IPython

IPython provides many useful tools for a user to quickly access information, such as, how to call a function? What arguments and options does a function have? What does the source code of a Python function look like? And so on.

In this session, you will use one such tool, namely the **?** character to explore documentation,.


- Accessing documentation with **?**

Python has a built-in **help()** function that can access the concise summary of each Python object. For example, to see the documentation of the built-in **len** function, you can type **help(len)** in an empty cell and then click the 'run' button :

```
help(len)
```

Help on built-in function len in module builtins:

```
len(obj, /)
    Return the number of items in a container.
```

Or type **len?** in an empty cell and then click the ‘run’ button  Run:

```
len?
```

```
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

Finally, if you are looking for a guide to the Python language itself, you may have a look at <https://github.com/jakevdp/WhirlwindTourOfPython>.

#### 4. Introduction to NumPy

**Numpy** (<https://numpy.org/>), Numerical Python, provides an efficient interface to store and operate on dense data buffers. More detailed tutorial on **NumPy** can be found here: <https://numpy.org/devdocs/user/quickstart.html>.

You can import **NumPy** using **np** as an alias and check the version by typing the following (Note that it is a double underscore sign you need to type rather than one.):

```
import numpy as np
np.__version__
```

To display **NumPy**’s built-in documentation, you can type: **np?** Further, to display all the contents of the **numpy** namespace, you can use the Tab button, that is, type **np** followed by a full-stop (.) character and the Tab key. A list should appear next to **np**. You may press the Down arrow from the keyboard to scroll down the list.

NumPy arrays contain values of a single type. These types include Boolean, Integer, Float and Complex number. Details can be viewed here: <https://docs.scipy.org/doc/numpy/user/basics.types.html>

##### 4.1 The basic of NumPy arrays

- To generate an array using **Numpy**, you may use **np.array**. For example,  

```
A = np.array([1, 3, 6, 10])
```
- Attributes of arrays: Determining the size, shape, memory consumption and data types of arrays.  
Type the following code to generate a one-dimensional and two-dimensional array in the **Jupyter** notebook:

```
import numpy as np
np.random.seed(421) # seed with a set value to ensure that the same random arrays are generated each time

a = np.random.randint(5, size=4) # One-dimensional array
b = np.random.randint(5, size=(2,3)) # Two-dimensional array
```

### Task1:

- 1) Understand the meaning of value 5 in the above code by typing `np.random.randint?`
- 2) Use the built-in **print** function to print out **a** and **b** arrays. If you do not know how to use **print**, do not forget to type **help(print)** or **print?** to look for help.

Each array has attributes **ndim** (the number of dimensions), **shape** (the size of each dimension), **size** (the total size of the array), **dtype** (the data type of the array), **itemsize** (the size in bytes of each array element) and **nbytes** (the total size in bytes of the array). You may use print to check these attributes of each array. For example,

```
print("a ndim: ", a.ndim)
```

**Task2:** Investigate these attributes of array **b** using the built-in function **print()**.

- Indexing of arrays: getting and setting the value of individual array elements.

Note that in Python, the index of an array counts from **zero**.

In a one-dimensional array, you can access the  $i^{th}$  value by specifying the desired index in **square brackets**. For example, the first element in **a**, that is **a[0]**. To index from the end of the array, you can use negative indices. For example, the last element in **a**, this is, **a[-1]**.

### Task3:

- 1) Print array **a**
- 2) Access the second value in **a**
- 3) Access the last second value in **a** using the negative index.

In a multidimensional array, you access items using a comma-separated tuple of indices. For example, **b[0,0]**, or **b[1,-1]**.

### Task4:

- 4) Print array **b**.
- 5) Access the last second value in the first row of **b**.
- 6) Access the second value in the second row of **b** using the negative indices.

- Slicing of arrays: getting and setting smaller subarrays within a larger array.

To access a slice of an array **x**, you can use **x[start:stop:step]**. The default values are **start=0, stop = size of dimension, step=1**.

**Task5:**

- 1) Generate an array **c** including 12 elements by typing  
**c = np.arange(12).**
- 2) Use the help function to find out more about **np.arange**.
- 3) Investigate on how to get subarrays by typing the following code (do not forget to press the **run** button to see the results):
  - a. **c[:4]**
  - b. **c[0:4:2]**
  - c. **c[::3]**
  - d. **c[1::2]**
  - e. **c[5::-1]**

Multidimensional slices work in the same way, with multiple slices separated by comas. For example,

```
x = np.random.randint(12, size=(4,3))
```

x

```
array([[ 7,  3,  8],
       [ 7,  6,  8],
       [ 0,  4, 10],
       [ 0, 10,  7]])
```

```
x[1:2:, 0:2]
```

```
array([[7, 6]])
```

- Reshaping of arrays: changing the shape of a given array

You can change the shape of a given array by using the **reshape()** method. For example,

```
d = c.reshape((3,4))
```

d

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

**Task6:** change the shape of array **b** from 2 by 3 to 3 by 2.

- Joining and splitting of arrays: Combining multiple arrays into one and splitting one array into many.

You can use **np.concatenate**, **np.vstack** and **np.hstack** to concatenate or join two arrays. The opposite of concatenation is splitting, which can be done by the functions **np.split**, **np.hsplit** and **np.vsplit**.

#### Task7:

- 1) Use the help function to find out more about **np.concatenate**, **np.vstack**, **np.hstack**, **np.split**, **np.hsplit** and **np.vsplit**.
- 2) Generate a two-dimensional array by typing

```
M = np.array([[10, 9, 8],[0, 1, 2]])
```

Use **np.concatenate** to get new arrays as shown as follows:

```
Q = np.concatenate([M, M], axis=1)
Q
array([[10, 9, 8, 10, 9, 8],
       [ 0, 1, 2,  0, 1, 2]])
```

How to get an array like the following one using **np.concatenate**?

```
array([[10, 9, 8],
       [ 0, 1, 2],
       [10, 9, 8],
       [ 0, 1, 2]])
```

- 3) How to split array `x1=[1, 2, 3, 41, 52, 8, 9, 10]` into three subarrays, which are `[1, 2, 3]`, `[41, 52]` and `[8, 9,10]` using **np.split**?

## 4.2 Using matrices in Python

### 4.2.1 How to define matrices

In this section, you will learn two ways to define a matrix.

- You can define a matrix using a pair of square brackets. Inside of it, each row is surrounded by one pair of square brackets. Elements in each row are separated by comma; rows in the matrix are also separated by comma. For example, type the following in an empty cell and then click the 'run' button

Run

```
L = [[1,2,3],[0,10,-5],[-6,-2,11]]
```

To access an element, you can give the corresponding index in its row and column. For example, if you want to access the element in the second row and the third column, you can type the following (As you have learnt that the index of an array counts from **zero**.)

```
print(L[1][2])
```

-5

Note that you cannot put two indices in one square as a tuple. For example,

```
L[1,2]
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-27-db2a5cd061ea> in <module>()  
----> 1 L[1,2]  
  
TypeError: list indices must be integers or slices, not tuple
```

- You can import **NumPy** using **np** as an alias, then use **np.array** to define a matrix. For example,

```
L1= np.array([[1, 2, 3],[0, 10, -5],[-6, -2,11]])
```

To access an element, you can do either

```
L1[1][2].
```

or

```
L1[1, 2].
```

- There is a class: **numpy.matrix**. However, it is not recommended to use this class, even for linear algebra. Therefore, we will not cover this class in our module.

4.2.2 Basic operation: you will use built-in functions in **NumPy** to do basic computation.

Type the following code in each block into your **jupyter** notebook and then click the ‘run’ button. To see what you get, you can either call the variable’s name directly or use the print function.

```
a = np.array([10, 15, -2, 9])  
b = np.arange(4)
```

```
c= a-b
```

```
a*2  
b**2
```

What is the difference between using ‘\*’ and ‘\*\*’?

```
M= np.array([[2, 3], [5, -8]])
```

```
N =np.array([[1, -4],[8, -6]])
```

```
M + N
```

```
M * N  
M @ N  
M.dot(N)
```

As you can see that '\*' returns the product of two matrices, element-wise; '@' and '.dot()' are used for matrix multiplication, which you will learn in the next lecture.

```
np.transpose(M)
```

```
np.trace(M)
```

The trace function is used to compute the sum of elements along the main diagonal line in the matrix.

```
np.diag(M)
```

The **diag** function is used to get all elements along the main diagonal line in the matrix.

### 4.2.3 numpy.linalg

**numpy.linalg** is a class having a standard set of things like inverse and determinant. matrix decompositions. You will learn the matrix decomposition in the next lecture. Please not to worry if you do not know these concepts yet. In this practical session, please focus on how to use the correct Python functions to solve each task. You need to import **numpy.linalg** first.

#### Task8:

- 1) compute the inverse of the matrix M defined in 4.2.2 using **linalg.inv(a)**.
- 2) compute the determinant of matrix N defined in 4.2.2 using **linalg.det(a)**.

Note that the way you call each function can be different depending on how you import the **numpy.linalg** module, that is, with or without an alias. More details can be found here:

<https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>

## 5 Computation on Numpy Arrays

### 5.1 Universal functions (UFuncs)

- Arithmetic operators implemented in NumPy

Operator	Equivalent ufunc	Description
+	<b>np.add</b>	Addition
-	<b>np.subtract</b>	Subtraction
-	<b>np.negative</b>	Unary negation (e.g. -10)
*	<b>np.multiply</b>	Multiplication
/	<b>np.divide</b>	Division
//	<b>np.floor_divide</b>	Floor division (e.g. 6//5=1 )
**	<b>np.power</b>	Exponentiation
%	<b>np.mod</b>	Modulus/remainder (e.g. 7%3=1)

- Absolute value: use **np.abs** or **np.absolute**

**Task9:** exploring NumPy's UFuncs. Generate an array **x2 = [-10, 2, 3,-5]**

Do the following:

- 1) `np.abs(x2)`
- 2) `x2+4`
- 3) `x2-3`
- 4) `-x2`
- 5) `x2 %2`

- Comparison operators as ufuncs

**Task10:** Your task is to understand the function of each operator listed in the following table by working on some your own examples, and then fill in the last column with the corresponding result. You may use **x2** generated previously as the input of each operator. For example, you may type **x2 >2** in the column of **Typing command** to obtain the result.

Operator	Equivalent ufunc	Typing command	Result
<code>==</code>	<code>np.equal</code>		
<code>!=</code>	<code>np.not_equal</code>		
<code>&lt;</code>	<code>np.less</code>		
<code>&lt;=</code>	<code>np.less_equal</code>		
<code>&gt;</code>	<code>np.greater</code>	For example: <b>x2 &gt;2</b>	
<code>&gt;=</code>	<code>np.greater_equal</code>		

## 5.2 Aggregations: Summing, Minimum and Maximum

NumPy provides many aggregation functions. You can see some of them in the following table. Quite often, you can see some missing values (may be denoted to NaN) in a given data file. Most aggregates have a NaN-safe counterpart that computes the result while ignoring missing values.

**Task11:** Your task is to understand the meaning of each function listed in the following table by working on some examples you want to try, and then fill in the last column with the corresponding result. You may use **x2** again as the input of each function you will practice. For example, you may type **np.sum(x2)** or you may type **x2.sum()** to obtain the result of the sum of **x2**.



Function Name	NaN-safe Version	Description	Result
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements	
<code>np.prod</code>	<code>np.nanprod</code>	Compute product of elements	
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value	
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value	
<code>np.argmin</code>	<code>np.nanargmin</code>	Find index of minimum value	
<code>np.argmax</code>	<code>np.nanargmax</code>	Find index of maximum value	
<code>np.any</code>	N/A	Evaluate whether any elements are true	
<code>np.all</code>	N/A	Evaluate whether all elements are true	

If you have an array with more than one dimension, you can use *axis* as an additional argument specifying along which dimension the aggregate is computed. **axis=0** specifies the aggregate is computed along column; **axis=1** is along each row. By default, each NumPy aggregation function will return the aggregate over the entire array.

### 5.3 Comparisons and Boolean logic

The following table summarizes the Boolean (true/false statements) operators and their equivalent ufuncs:

Operator	Equivalent ufunc	Typing command	Result
<code>&amp;</code>	<code>np.bitwise_and</code>		
<code> </code>	<code>np.bitwise_or</code>		
<code>^</code>	<code>np.bitwise_xor</code>		
<code>~</code>	<code>np.bitwise_not</code>		

Using these tools, you can manipulate values in an array based on some criterion. For example, generate a new array **x3** by typing **x3 = np.arange(15)** first. To get the sum of elements whose value is either greater than 10 or less than 3, you can type the following:

```

sx3= np.sum(x3[(x3>10) | (x3<3)])

```

#### Task12:

Compute the product of elements in **x3** whose value is in between 3 and 10 inclusive using the proper equivalent ufuncs from NumPy.

### 5.4 Selecting random points: sometimes you need to randomly select a number of data points from a dataset.

For example, generate an array as follows:

```

rand=np.random.RandomState(421)
rx = rand.randint(50, size=10)
print(rx)

```

```
[23 44 19 17 15 30 38 17 47 32]
```

Suppose we want to randomly select 5 data points from this array. You can type the following code:

```

indices = np.random.choice(rx.shape[0], 5, replace=False)
indices

```

```
array([5, 2, 8, 4, 7])
```

```

selected = rx[indices]
print(selected)

```

```
[30 19 47 15 17]
```

#### Task13:

Can you modify the value of those 5 selected elements to zero?

#### 5.5 Sorting arrays

Although Python has built-in **sort** and **sorted** functions to work with lists, NumPy provide **np.sort** and **np.argsort**, which are much more efficient and useful due to using a *quick-sort* algorithm.

**np.sort**: return a sorted version of the array

**np.argsort**: return the indices of the sorted elements.

**Task14**: investigate how to use these two functions by using the help function first. Then apply these two functions on x4=[10, 9, -1, 3]. Further, can you sort x4 in descending order?

## 6 Visualisation with Matplotlib

**Matplotlib** (<http://matplotlib.org>) is a multiplatform data visualisation library built on **NumPy** arrays. You can import what you need for your visualisation purposes with the following command:

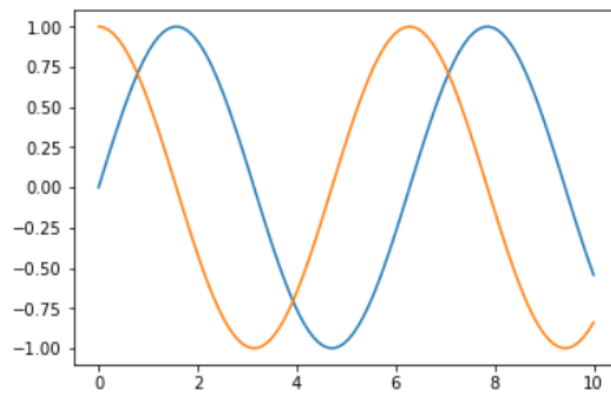
```
import matplotlib.pyplot as plt
```

The following shows a simple example, where **np.sin()** and **np.cos()** are trigonometric functions, which you will see again in Practical 3 :

```
: import matplotlib.pyplot as plt  
import numpy as np
```

```
: x = np.linspace(0, 10, 100)
```

```
: plt.plot(x, np.sin(x))  
plt.plot(x, np.cos(x))  
  
plt.show()
```



## References

1. J. VanderPlas: Python Data Science Handbook, 2016, O'Reilly Media, Inc.