CSC2014: Lab 2 – NumPy and Array Manipulation

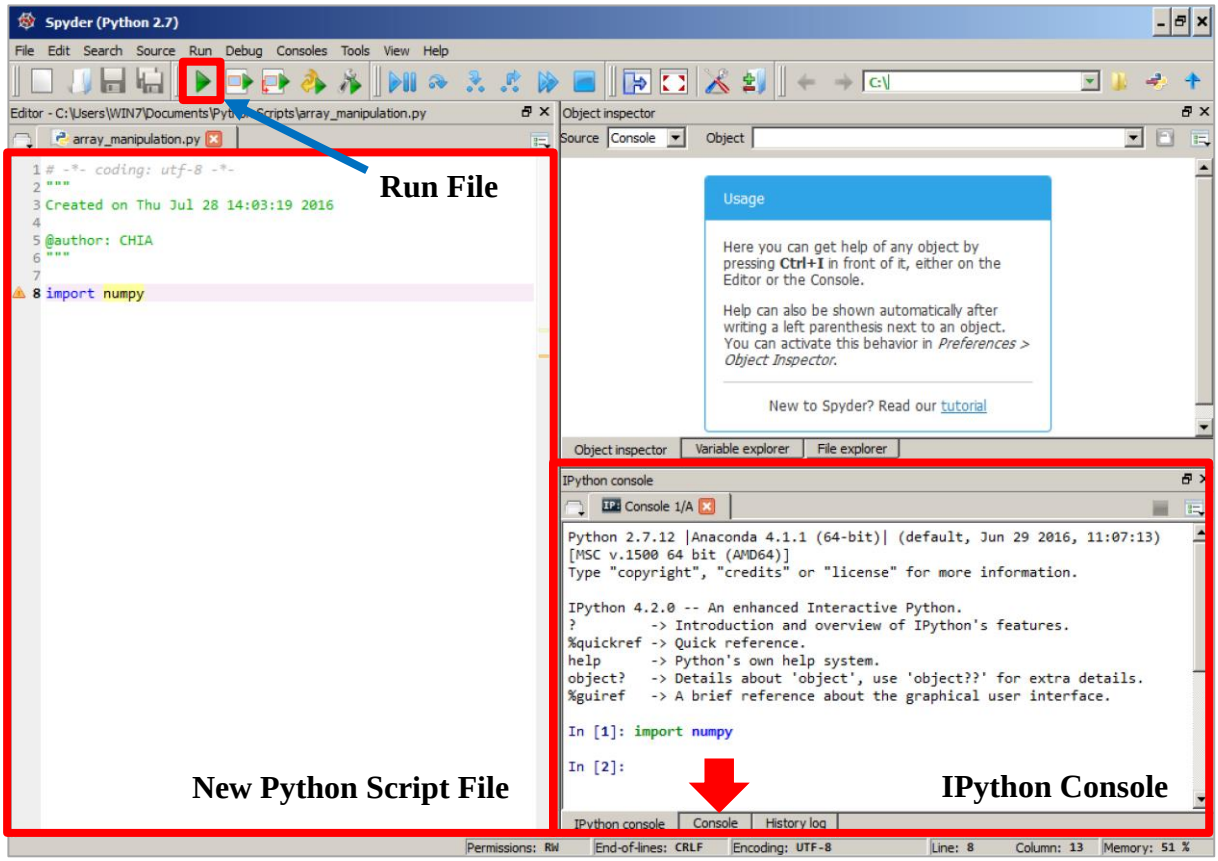
A. NumPy

NumPy is a package for Python that provides support for multi-dimensional arrays. It allows images to be expressed as multi-dimensional arrays and serves as the building block of many other packages like OpenCV.

First, we have to import the package by using the following command. You can choose to do it in the IPython console, or put the command into a new Python script file and click the **Run File** button to run it later. If you choose the latter approach, switch from the IPython console to the normal console. The IPython console is more for interactive programming instead of batch execution.

```
import numpy
```

Important
Remember to import this package because we will be dealing with arrays most of the time.



B. Creating an Array using NumPy in Spyder

The easiest way to create a small array is to manually enter the numbers by using the **`array`** function in NumPy. Entering the following three commands will create a 1x3 integer array, a 3x3 integer array, and a 1x3 floating-point array respectively.

Important
Take note on the comma location and the number of round and square brackets.

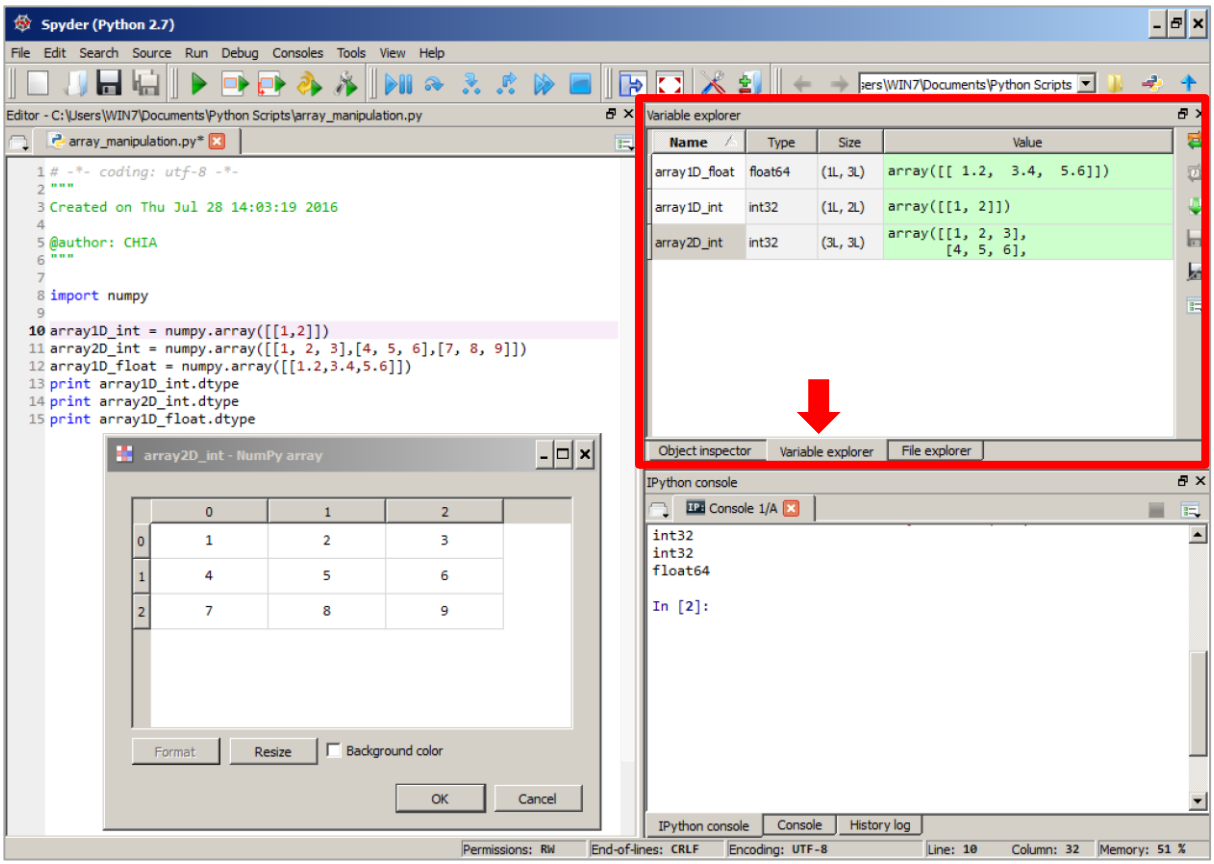
```
array1D_int = numpy.array([[1,2]])                    #Create a 1x2 array
array2D_int = numpy.array([[1,2,3],[4,5,6],[7,8,9]])  #Create a 3x3 array
array1D_float = numpy.array([[1.2,3.4,5.6]])          #Create a 1x3 array
```

After creating the arrays, we may check the numbers as well as the arrangement of each by printing the array using the **`print`** function.

Python 3.x Version

```
print (array1D_int)
print (array2D_int)
print (array1D_float)
```

Alternatively, we can also check the arrays by using the Variable Explorer. Double-click on the array name will open a window that shows the values as well as how they are stored.



Exercise 1

Create the following three arrays by using the **`array`** function. Save your Python script file as **`lab2_ex1.py`**.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

4x4

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

5x4

1
2
3
4

4x1

The **`array`** function takes two arguments, (i) the numbers to be converted into an array and, (ii) the data type of each element. If we never specify the data type, the **`array`** function will automatically choose the most suitable data type, based on the numbers that we entered. We may check the data type by using the **`dtype`** function.

Python 3.x Version

```
print (array1D_int.dtype)                    #int32
print (array2D_int.dtype)                    #int32
print (array1D_float.dtype)                  #float64
```

If we would like to choose our own data type, then we should include the second argument that specifies the **`dtype`** of an array. The following two commands create a 1x2 8-bit unsigned integer array and a 2x3 64-bit floating-point array respectively.

```
array1D_uint8 = numpy.array([[1,2]],dtype=numpy.uint8)
array2D_float = numpy.array([[1,2,3],[4,5,6]],dtype=numpy.float64)
```

Data types that will be commonly used in this subject are shown in the following table.

Data Type	Description
`int8`	Integer number ranging from -128 to 127.
`int32`	Integer number ranging from -2147483648 to 2147483647.
`uint8`	Unsigned integer number ranging from 0 to 255.
`float64`	Double precision floating-point number.
`complex128`	Complex number represented by two 64-bit floats (real and imaginary).

Challenge 1
Enter the following command in the IPython console and observe the values inside the array. Try to understand the reasons of not getting the three values that you have entered.

```
array1D_uint8 = numpy.array([[255,256,257]],dtype=numpy.uint8)
```

Hint
If you think typing **`numpy`** over and over again when calling the functions is very tedious, you may choose to import the package under a shorter name like **`np`**.

```
import numpy as np
array1D_int = np.array([[1,2,3]])
array1D_float = np.array([[1.2,3.4,5.6]],dtype=np.float64)
```

C. Accessing Elements in an Array

After creating an array, the elements of the array can be modified or extracted. In image processing, this is equivalent to changing or extracting the intensity values.

Important
Remember the coordinate is always expresses in the form of (row, column), and Python uses zero-based indexing (index starts at zero).

```
array2D_int[0,0] = 15       #Change the value at [0,0] to 15
array2D_int[1,2] = 20       #Change the value at [1,2] to 20
```

```
e2_2 = array2D_int[2,2]     #Extract the value at [2,2] to e2_2
e1_1 = array2D_int[1,1]     #Extract the value at [1,1] to e1_1
```

If we would like to change or extract a set of values, then we have to make use of the colon operator (**:**). This is applied to create a range of numbers that can in turn be used to access the array.

```
array2D_int[1,0:2] = 99     #Change the value at [1,0] and [1,1] to 99
array2D_int[2,0:3] = 20     #Change the value from [2,0] to [2,2] to 100
e2nd = array2D_int[1,1:3]   #Extract the value at [1,1] and [1,2] to e2nd
e3rd = array2D_int[2,0:3]   #Extract the value from [2,0] to [2,2] to e3rd
```

Important

The colon operator (**:**) can have three arguments, and they are written in the form of **`start_index : end_index : step`**. If the last argument is omitted, it will take the default value of one. Please take note that the **`end_index`** value is **not** included in the range.

When we are extracting a set of values, all the values will be placed inside an array. Hence in the previous example, both **`e2nd`** and **`e3rd`** are arrays. If we need to split the values into separate variables, then we can use the following command. In this case, the three extracted values will be saved into variable **`e2_0`**, **`e2_1`**, and **`e2_2`** respectively. This is also applicable when we need to unpack a set of values returned from a function.

```
[e2_0,e2_1,e2_2] = array2D_int[2,0:3]
```

Exercise 2

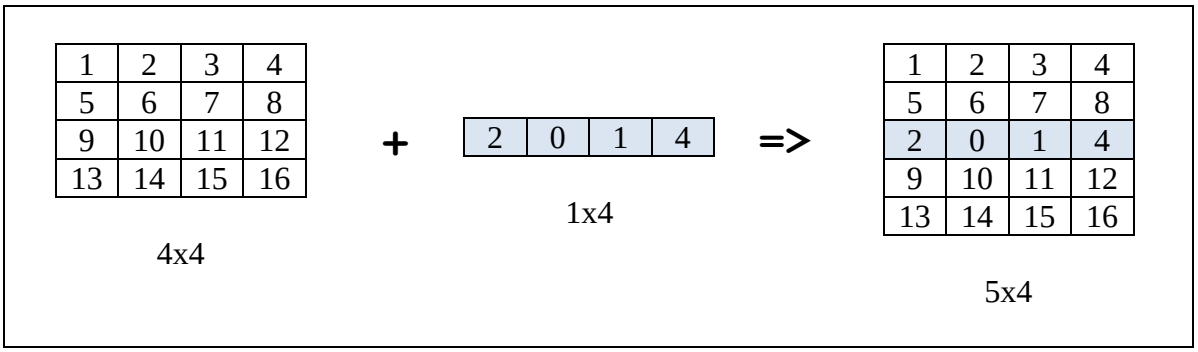
Change all the elements in the first column of **`array2D_int`** to 2014. Save your Python script file as **`lab2_ex2.py`**.

D. Concatenation of Arrays

Arrays can be concatenated (join together) to create a larger array. This can be achieved by using the **`concatenate`** function in NumPy. It takes two arguments. The first argument is the two arrays to be concatenated (in round brackets), and the second argument is the concatenation axis. The concatenation can be carried out vertically (0) or horizontally (1).

```
arr2D_1 = np.array([[1,2],[3,4]])               #Create a 2x2 array
arr2D_2 = np.array([[5,6],[7,8]])               #Create a 2x2 array
vert_arr = np.concatenate((arr2D_1,arr2D_2),0)  #Vertical Concatenation
horz_arr = np.concatenate((arr2D_1,arr2D_2),1)  #Horizontal
                                                Concatenation
print vert_arr.shape                            #Check the array
print horz_arr.shape                            #Check the array
```

Challenge 2
Given a 4x4 array (leftmost), use the concatenation function to create a new 5x4 array that includes an additional row of numbers (rightmost).



E. Creating Large Arrays

If we only need to create a small array, it is fine to use the **`array`** function and manually enter all the numbers. But when we need to create larger arrays, we should create them by using the built-in functions like **`zeros`** and **`ones`**.

```
big_arr0 = np.zeros((100,100),dtype=np.uint8)    #100x100 array of zero
big_arr1 = np.ones((100,100),dtype=np.int32)     #100x100 array of one
big_arr2 = np.zeros((100,100,3),dtype=np.uint8)  #100x100x3 array of zero
big_arr3 = np.ones((100,100,8),dtype=np.int32)   #100x100x8 array of one
```

Exercise 3
Create a 200x200 array filled with the value of 100. Save your Python script file as **`lab2_ex3.py`**.

The above functions are sufficient to create any large arrays that carry the same values. However, if we need the values to follow a particular sequence, then we have to use the for-loop.

```
big_arrn = np.zeros((1,10),dtype=np.uint8)   #1x10 array of zero
for y in range(0,10,1):                      #Iterate y from 0 to 9
      big_arrn[0,y] = y                      #Save y into location (0,y)
```

In the previous example, an one-dimensional 1x10 array is created. Then a for-loop that iterates **`y`** over the range of 0 to 9 is created. Similar to the colon operator, the **`range`** function can have three arguments, and they are written in the form of **`(start_index, end_index, step)`**. If the last argument is omitted, then it will take the default value of one. Basically, the above for-loop is used to loop through all the columns of the array, and fill each column with the value of **`y`**.

Important
Remember to include a colon operator (**`:`**) at the end of a for-loop statement.

However, if we are dealing with a two-dimensional array, then we need to have two for-loop, one to iterate over all the columns, and one for all the rows.

```
big_arrn = np.zeros((10,10),dtype=np.uint8)  #10x10 array of zero
for x in range(0,10):                        #Iterate x from 0 to 9
      for y in range(0,10):                  #Iterate y from 0 to 9
            big_arrn[x,y] = y                #Save y into location (x,y)
```

Challenge 3

Create the following array by using the **`zeros`** function and for-loop.

1	0	2	0	3	0	4
0	0	0	0	0	0	0
5	0	6	0	7	0	8
0	0	0	0	0	0	0
9	0	10	0	11	0	12
0	0	0	0	0	0	0
13	0	14	0	15	0	16

7x7

-- END --