By Dipendra Shrestha

NumPy Tutorial

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

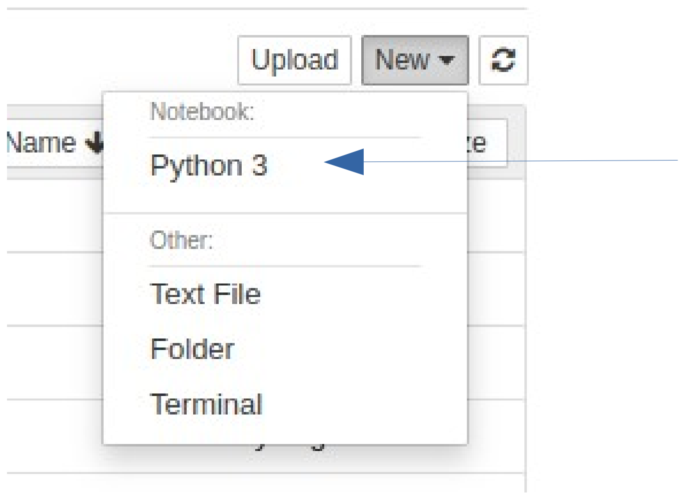
To install Numpy we use pip command as:-
→ **pip install numpy**

To install Jupyter we use pip command as:
→ **pip install Jupyter**

After installing the jupyter to run the jupyter we give command as:
→ **jupyter notebook**

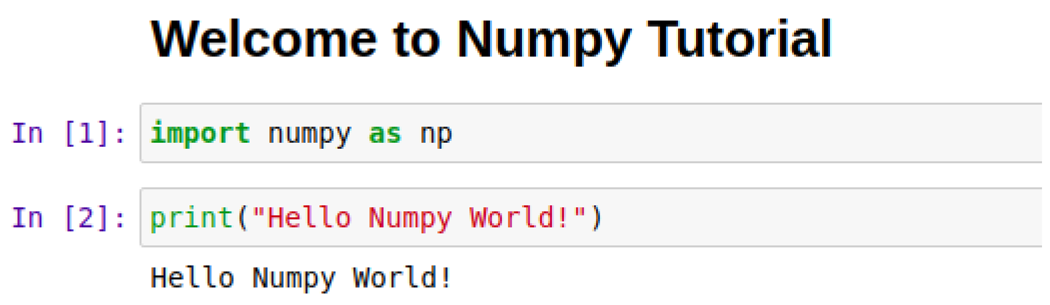
1. Create First Program in Jupyter

Now,
We create a new Notebook with python 3 as shown below:



Create a Notebook with Python 3

→ First program on Jupyter Notebook
- Each shell can run code independently.
- To run the code within each shell press **shift + enter.**



Welcome to Numpy Tutorial

```
In [1]: import numpy as np

In [2]: print("Hello Numpy World!")

        Hello Numpy World!
```

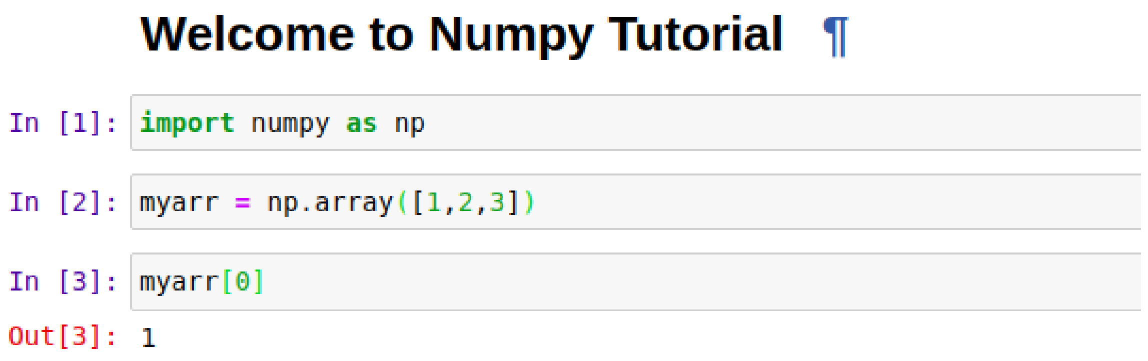
Here, in above code the shell that runs one after another are indexed by Jupyter itself as shown in In [1] : , In[2]: and so on.

By Dipendra Shrestha

2. Array Concept

→ Simple one dimension array

- We create a simple one dimensional array in



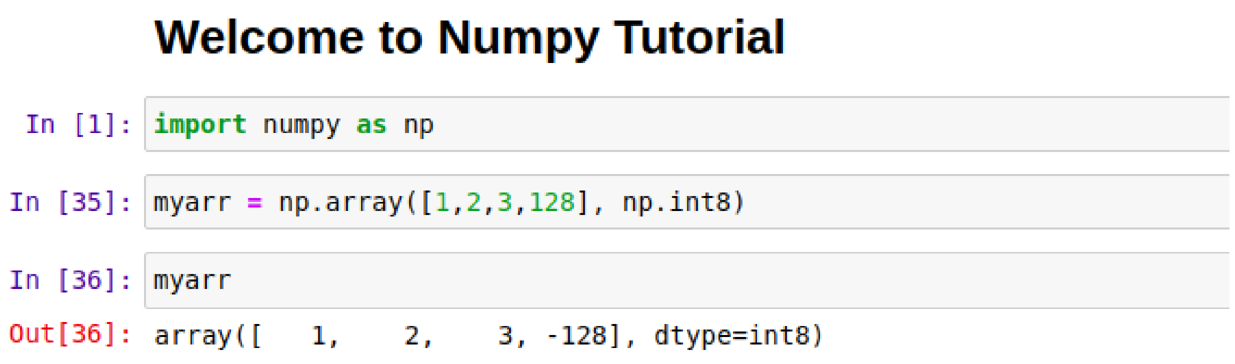
Welcome to Numpy Tutorial ¶

```
In [1]: import numpy as np

In [2]: myarr = np.array([1,2,3])

In [3]: myarr[0]
Out[3]: 1
```

a. At first ‘numpy’ is imported as ‘np’
b. Array with the name ‘myarr’ is created
c. As array index starts with 0 so if we execute myarr[0] it gives ‘1’ as a result.

myarr[0] → 1
myarr[1] → 2
myarr[2] → 3

- We can easily maintain the memory and specify how much memory should be consumed by the array.
- As we can specify the size of the integer of an array as ‘**np.int8**’ means that we are using 4-bit integer type.
- If the integer exceeds the 8-bit then we get the error as shown below



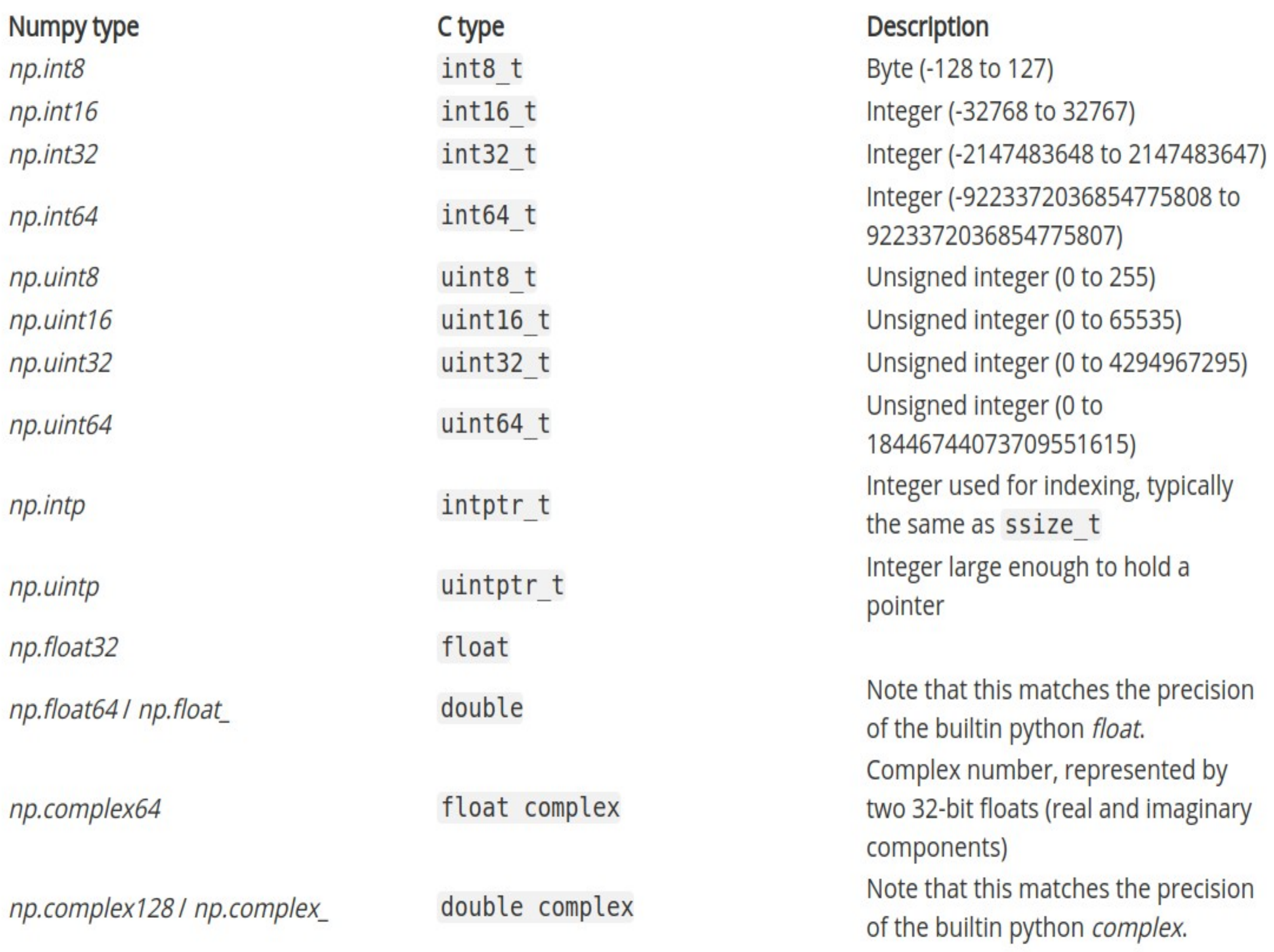
Welcome to Numpy Tutorial

```
In [1]: import numpy as np

In [35]: myarr = np.array([1,2,3,128], np.int8)

In [36]: myarr
Out[36]: array([   1,    2,    3, -128], dtype=int8)
```

↑

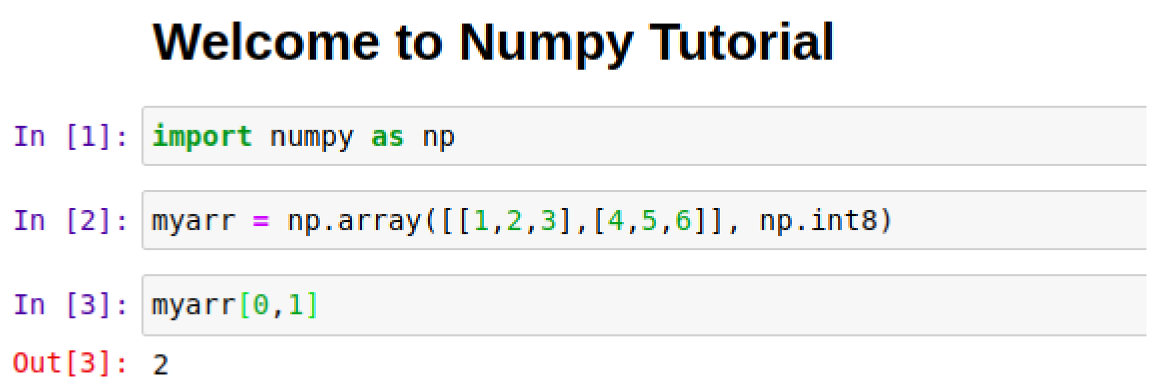
As int8 can only represent from
-128 to 127. So, if ‘128’ is placed
it shows in ‘-ve’

- A set of fixed-size aliases are provided as below:



Numpy type	C type	Description
np.int8	`int8_t`	Byte (-128 to 127)
np.int16	`int16_t`	Integer (-32768 to 32767)
np.int32	`int32_t`	Integer (-2147483648 to 2147483647)
np.int64	`int64_t`	Integer (-9223372036854775808 to 9223372036854775807)
np.uint8	`uint8_t`	Unsigned integer (0 to 255)
np.uint16	`uint16_t`	Unsigned integer (0 to 65535)
np.uint32	`uint32_t`	Unsigned integer (0 to 4294967295)
np.uint64	`uint64_t`	Unsigned integer (0 to 18446744073709551615)
np.intp	`intptr_t`	Integer used for indexing, typically the same as `ssize_t`
np.uintp	`uintptr_t`	Integer large enough to hold a pointer
np.float32	`float`	
np.float64 / *np.float_*	`double`	Note that this matches the precision of the builtin python *float*.
np.complex64	`float complex`	Complex number, represented by two 32-bit floats (real and imaginary components)
np.complex128 / *np.complex_*	`double complex`	Note that this matches the precision of the builtin python *complex*.

→ Two Dimensional Array



Welcome to Numpy Tutorial

```
In [1]: import numpy as np

In [2]: myarr = np.array([[1,2,3],[4,5,6]], np.int8)

In [3]: myarr[0,1]
Out[3]: 2
```

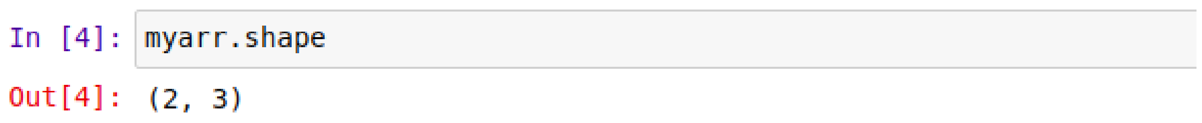
In two dimensional array concept values above in ‘ myarr ’ are arranged as :

[1,2,3] → 0th row
[4,5,6] → 1st row

value 1 => 0th row 0th column
value 2 => 0th row 1st column
value3 = > 0th row 2nd column

value4 => 1st row 0th column
value5 => 1st row 1st column
value6 => 1st row 2nd column

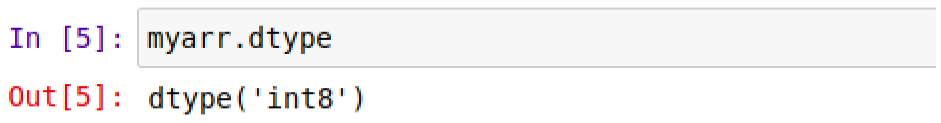
- In order to **know the structure of array** we can use 'array_name.shape' as:



```
In [4]: myarr.shape
Out[4]: (2, 3)
```

This shows that our array is named with 'myarr' and the structure of array is **(2,3)** which means that array has 2 rows and 3 column.

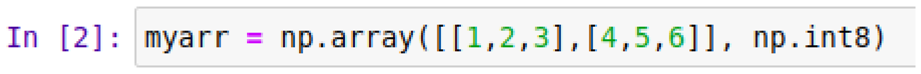
- In order to know the data type of the array we can use 'dtype'.



```
In [5]: myarr.dtype
Out[5]: dtype('int8')
```

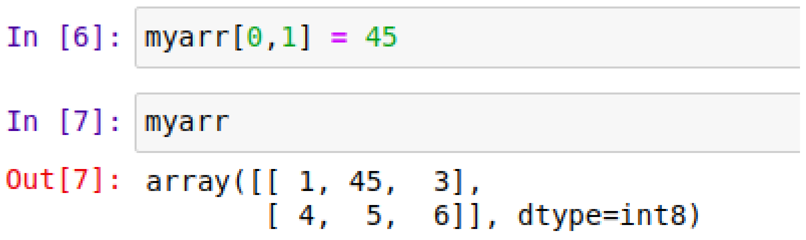
3. Changing the element of Array

Initially we we have following elements within a array.



```
In [2]: myarr = np.array([[1,2,3],[4,5,6]], np.int8)
```

Now if we want to change the element of the array we can simply change it as:



```
In [6]: myarr[0,1] = 45

In [7]: myarr
Out[7]: array([[ 1, 45,  3],
               [ 4,  5,  6]], dtype=int8)
```

As we can see that to change the element of array we assign the values to the particular position of the array.

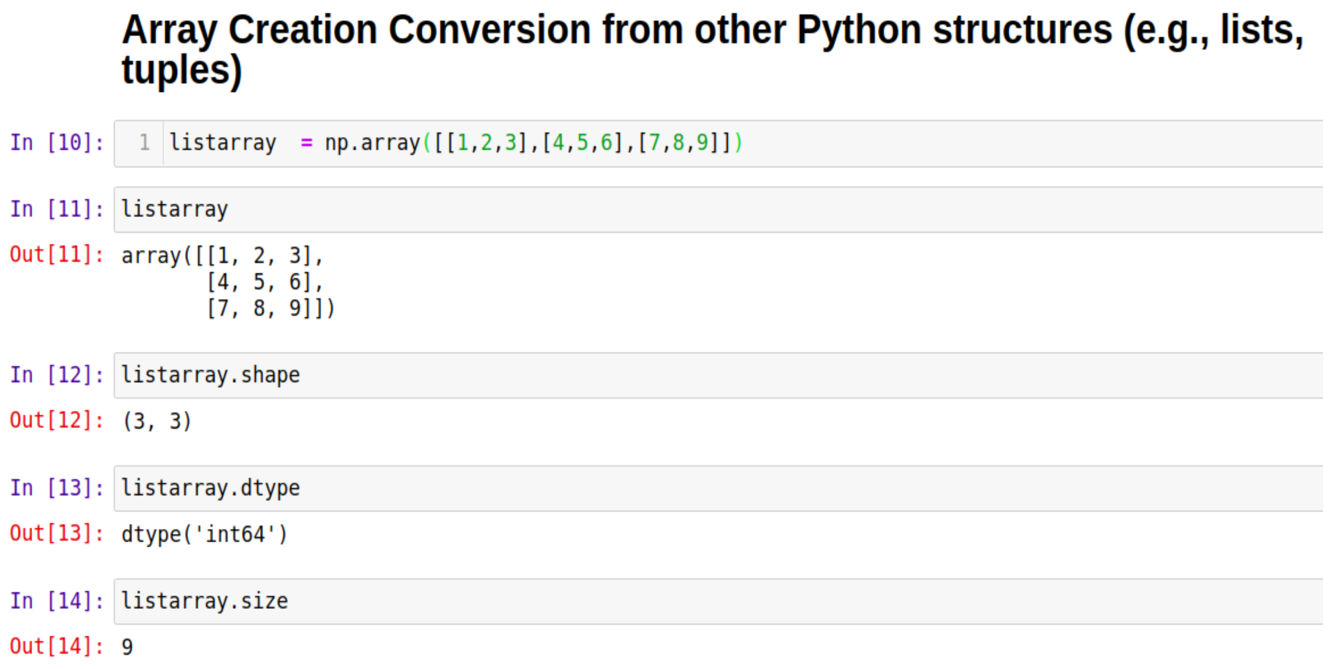
By Dipendra Shrestha

Array Creation in Numpy

There are five general mechanism for creating arrays in numpy.

1. Conversion from other Python structures (e.g., lists, tuples)
2. Intrinsic numpy array creation objects (e.g., arange, ones, zeros, etc.)
3. Reading arrays from disk, either from standard or custom formats
4. Creating arrays from raw bytes through the use of strings or buffers
5. Use of special library functions (e.g., random).

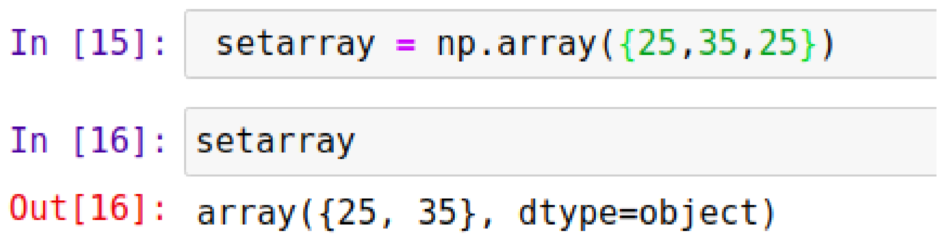
1. Conversion from other Python structures (e.g., lists, tuples)



Array Creation Conversion from other Python structures (e.g., lists, tuples)

```
In [10]: listarray  = np.array([[1,2,3],[4,5,6],[7,8,9]])

In [11]: listarray
Out[11]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [12]: listarray.shape
Out[12]: (3, 3)

In [13]: listarray.dtype
Out[13]: dtype('int64')

In [14]: listarray.size
Out[14]: 9
```

- We can create array using python structure such as list, tuples and sets.
- We can even make array using set as shown below but it’s not the efficient method as the ‘dtype’ of the array will be of object so we only use int or float type array for proper manipulation of data.

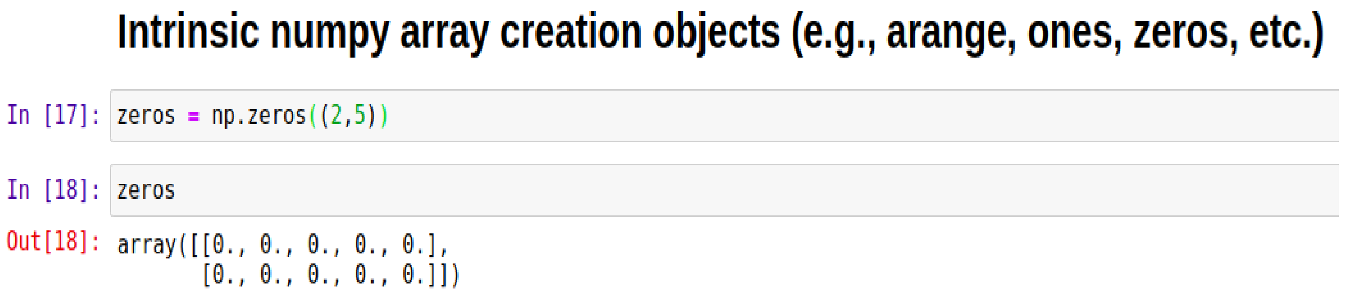


```
In [15]: setarray = np.array({25,35,25})

In [16]: setarray
Out[16]: array({25, 35}, dtype=object)
```

***Note:** set automatically removes the duplication of the elements within an array.*

2. Intrinsic numpy array creation objects (e.g., arange, ones, zeros, etc.)

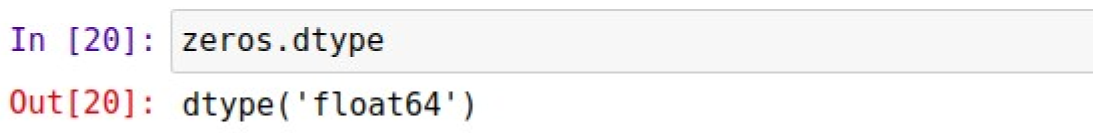
- ***zeros*** array creation



Intrinsic numpy array creation objects (e.g., arange, ones, zeros, etc.)

```
In [17]: zeros = np.zeros((2,5))

In [18]: zeros
Out[18]: array([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

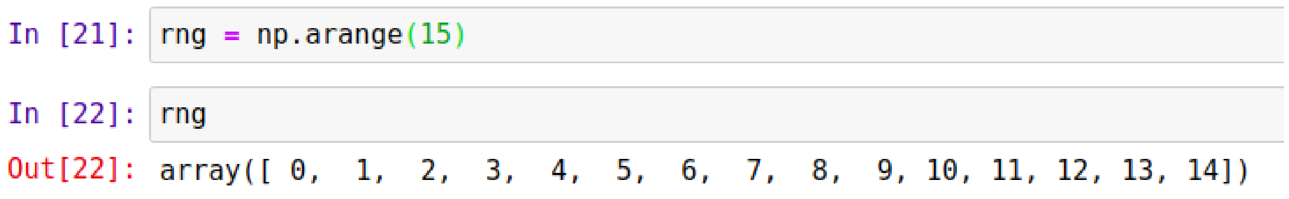
- The data type of the 'zeros' array is float64.



```
In [20]: zeros.dtype
Out[20]: dtype('float64')
```

- ***arange*** array creation

This '**arange'** function gives the array with 0 to n-1 numpy array.

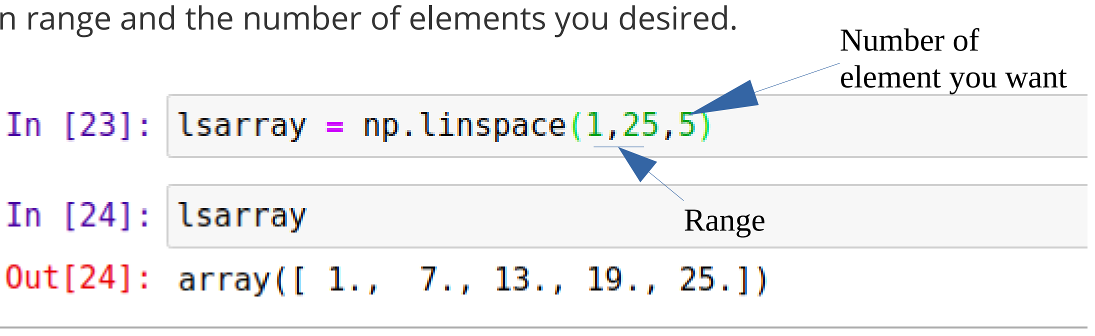


```
In [21]: rng = np.arange(15)

In [22]: rng
Out[22]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Using '**arange**' array we can simply loop it and access the elements of an array.

- ***linspace*** array creation

This '**linspace'** array provides equally linearly spaced array elements within a given range and the number of elements you desired.

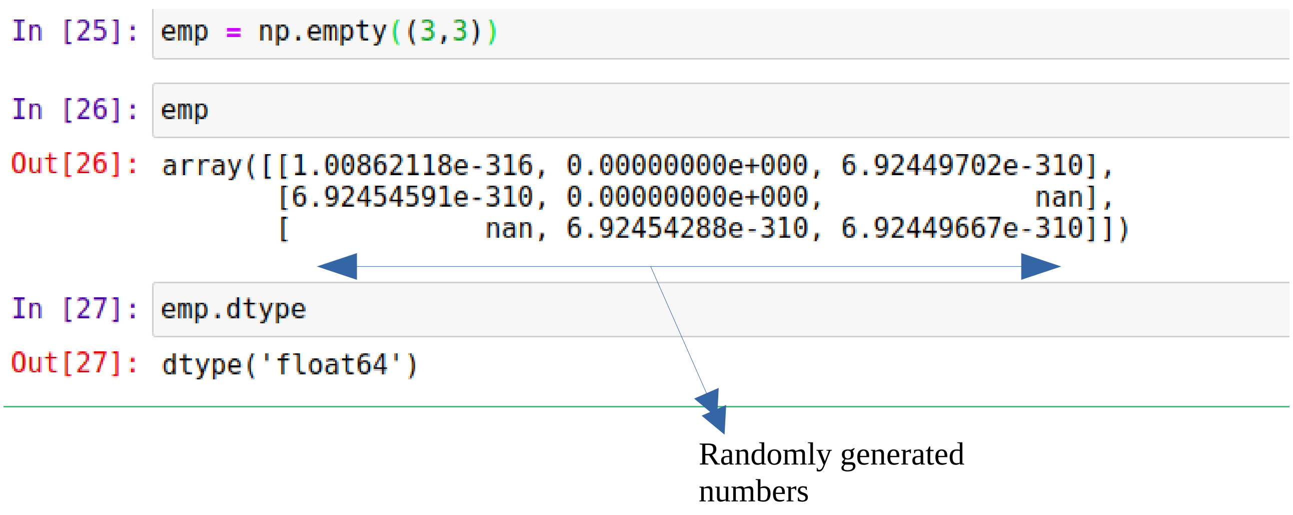


```
In [23]: lsarray = np.linspace(1,25,5)

In [24]: lsarray
Out[24]: array([ 1.,  7., 13., 19., 25.])
```

As we can see that elements that are generated has equal gap between each of the elements of the array. The element starts with '1' , ends at '25' and there are altogether '5' numbers of elements.

- ***empty*** array creation

This array generates the array as per the given size and the elements within that array are randomly generated float type numbers.

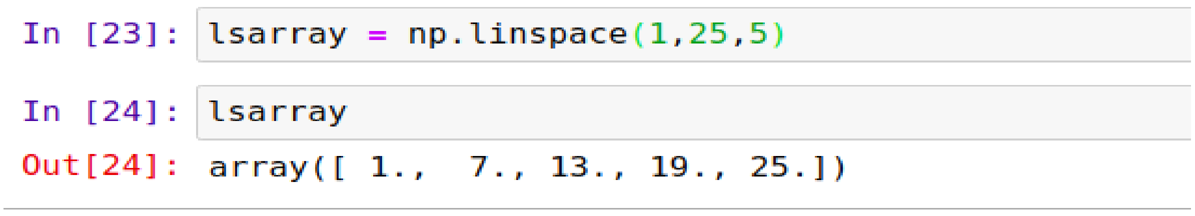


```
In [25]: emp = np.empty((3,3))

In [26]: emp

Out[26]: array([[1.00862118e-316, 0.00000000e+000, 6.92449702e-310],
                [6.92454591e-310, 0.00000000e+000,             nan],
                [            nan, 6.92454288e-310, 6.92449667e-310]])

In [27]: emp.dtype

Out[27]: dtype('float64')
```

- ***empty_like*** array creation

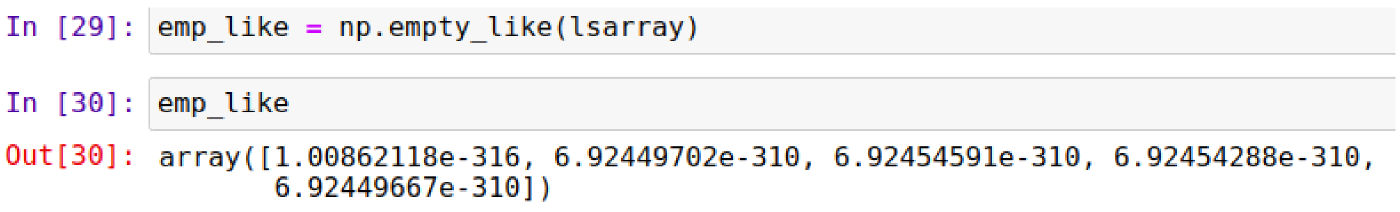
This **'empty_like'** array takes another array that has been already created and it takes the size of that array. As a result it gives empty array with the size of the array that it has taken.

Our previous ***'lsarray'*** *was as shown below:*



```
In [23]: lsarray = np.linspace(1,25,5)

In [24]: lsarray

Out[24]: array([ 1.,  7., 13., 19., 25.])
```

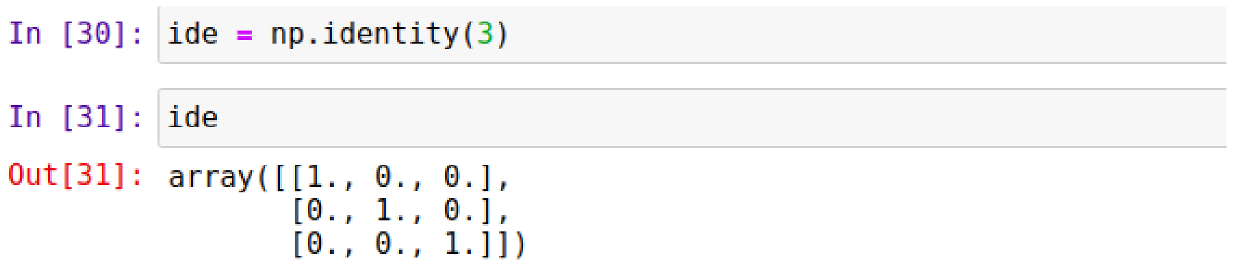
After passing the ***'lsarray'*** to the ***'empty_like'*** array we get the result as the empty array with the size of **'lsarray'**. Basically this concept of array concept are useful for efficiency and we can even easily modify the element of an array.



```
In [29]: emp_like = np.empty_like(lsarray)

In [30]: emp_like

Out[30]: array([1.00862118e-316, 6.92449702e-310, 6.92454591e-310, 6.92454288e-310,
                6.92449667e-310])
```

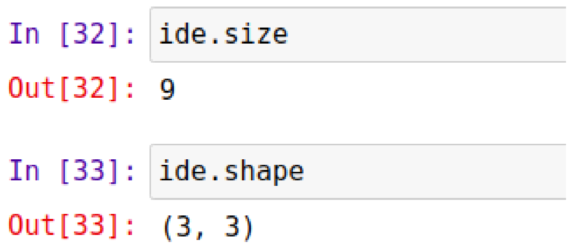
- ***Identity** array creation*

  This array gives the identity matrix as per the user gives the size of the identity matrix. Here, ***np.identity(3)*** *means that create an identity array of size 3 x 3.*



```
In [30]: ide = np.identity(3)

In [31]: ide
Out[31]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

As we can see that the size of the 'ide' which is identity array is 9 and the shape of the array is 3 x 3.



```
In [32]: ide.size
Out[32]: 9

In [33]: ide.shape
Out[33]: (3, 3)
```

By Dipendra Shrestha

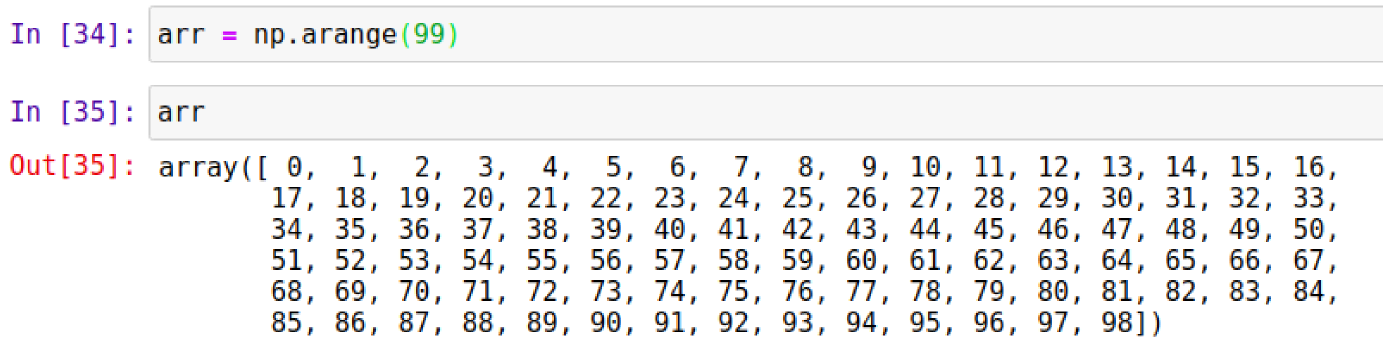
Two Important functions of numpy array:

1. reshape( )
2. ravel( )

1. reshape( )

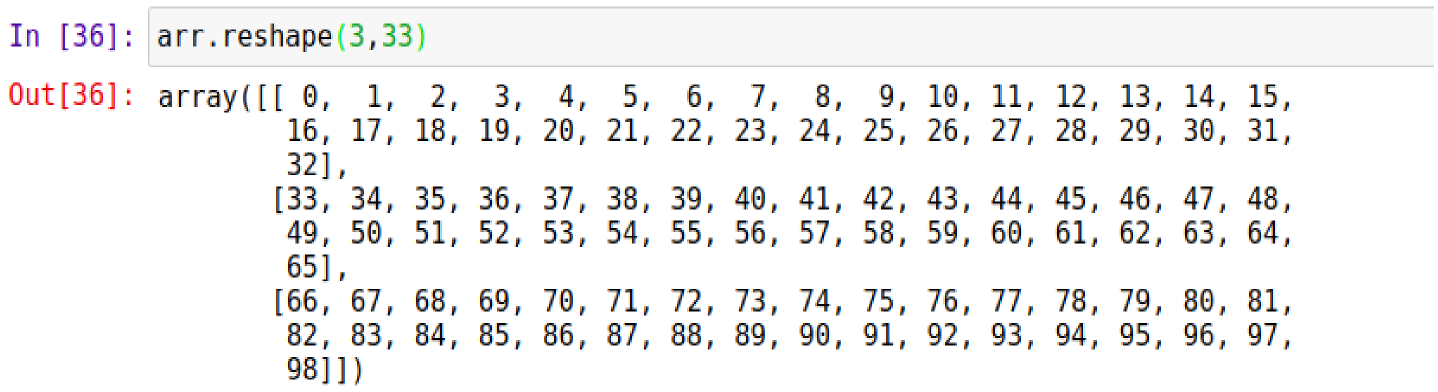
This function helps to reshape the array.

Suppose, If we use ***'arange'*** array and then generate the array which have elements from 0 to 98 I.e all together 99 elements as shown below .



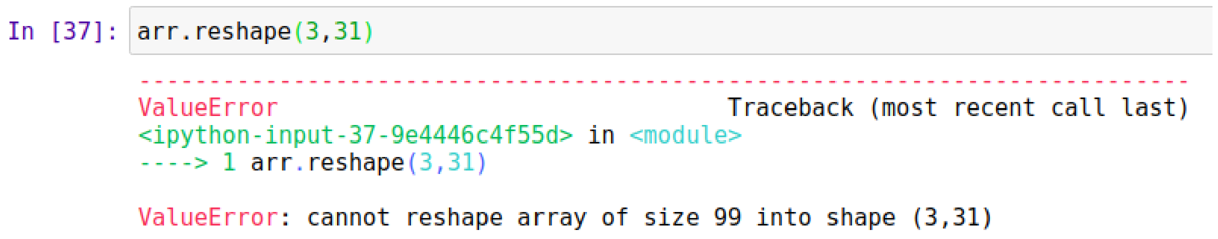
```
In [34]: arr = np.arange(99)

In [35]: arr

Out[35]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
                17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
                34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
                51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
                68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
                85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98])
```

Now, if we want to reshape the elements with in a 3 rows each row containing the 33 elements then we can simply use ***reshape( )*** function as reshape(3, 33).



```
In [36]: arr.reshape(3,33)

Out[36]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
                 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
                 32],
                [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
                 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
                 65],
                [66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
                 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
                 98]])
```

Note: In above listed example we can't give the size of reshape as **reshape (3,31)** as the reshape size must be exact multiple of the actual size of the array. As (3,31) can only contain the 91 elements but the actual elements of the array has to be 99 . So, this is consider as the invalid reshape.



```
In [37]: arr.reshape(3,31)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-37-9e4446c4f55d> in <module>
----> 1 arr.reshape(3,31)

ValueError: cannot reshape array of size 99 into shape (3,31)
```

Till now **'arr'** array hasn't been assigned with the reshaped array. So, inorder to get the reshaped array in our **'arr'** array we have to assign the reshaped array to **'arr'** array.

```
In [38]: arr = arr.reshape(3,33)

In [39]: arr

Out[39]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
                 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
                 32],
                [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
                 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
                 65],
                [66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
                 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
                 98]])
```

As we can see that our **'arr'** array has been updated.

2. ravel( )

This function helps to convert the reshaped array back to the one dimension array.
As in our case the reshaped array is **'arr'** and if we want to convert the reshaped array back to one dimensional array then simply we can use **ravel( )** function as shown below.

```
In [40]: arr = arr.ravel()

In [41]: arr

Out[41]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
                17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
                34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
                51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
                68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
                85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98])

In [42]: arr.shape

Out[42]: (99,)
```

As we can see that the **'ravel( )'** function has changed the reshaped array **'arr'** into the one dimensional array with 99 elements arranged in a single row.

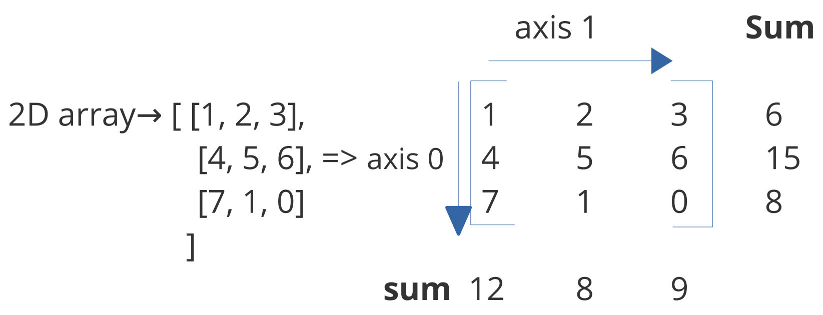
By Dipendra Shrestha

Numpy Axis

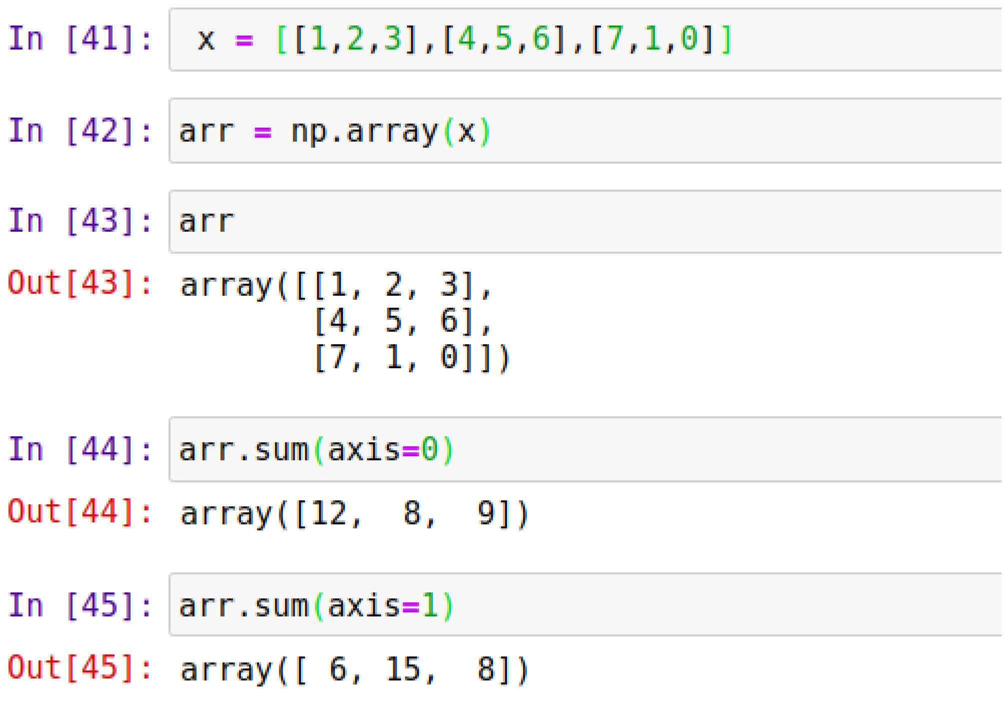
NumPy axis are the directions along the rows and columns. Just like coordinate systems, **NumPy** arrays also have **axes**. In a 2-dimensional **NumPy** array, the **axes** are the directions along the rows and columns.

In one dimension array there is only 1 axis as axis-0 but in case of two dimensional array it has two axis as axis-0 and axis-1. Axis of the array depends upon the dimension of an array. As the dimension of an array increases the axis also increases.

1D array → [1, 2, 3, 4, 5, 6] => axis-0



The sum of axis-0 is within a row I.e (1 + 4 + 7 = 12) and the sum of axis-1 is along the column I.e (1+ 2 + 3 = 6).



```
In [41]:  x = [[1,2,3],[4,5,6],[7,1,0]]

In [42]: arr = np.array(x)

In [43]: arr

Out[43]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [44]: arr.sum(axis=0)

Out[44]: array([12,  8,  9])

In [45]: arr.sum(axis=1)

Out[45]: array([ 6, 15,  8])
```

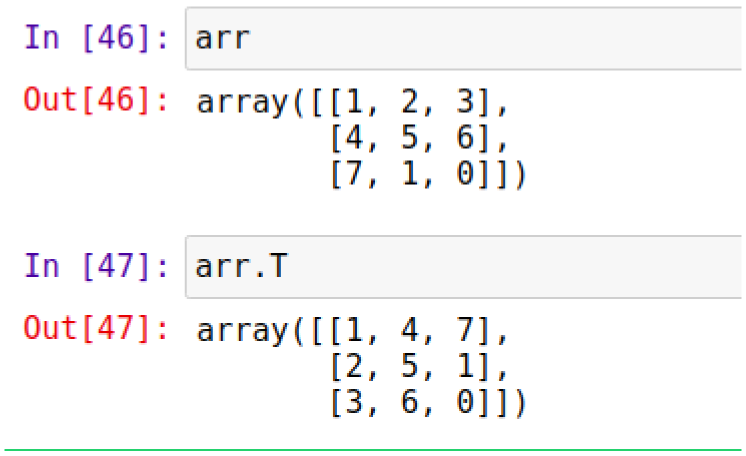
In above example we created one array with the name **'arr'.** Now, if we calculate the sum of the array along with the axis-0 then it manipulates the sum along with the rows whereas if we generate the sum along with the axis-1 then it manipulates the sum along with the column.

Some widely used numPy attributes and methods

=> Attributes

- **Transpose attribute**

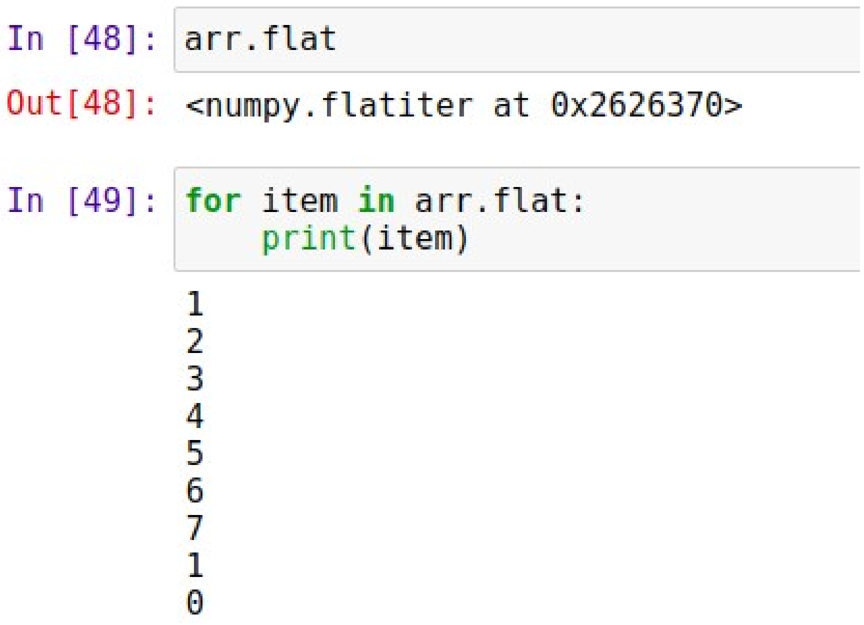
  Transpose attribute are used in numPy matrix to transpose the given matrix. Using this attribute we can easily interchange the rows elements to the column and column elements into the rows.



```
In [46]: arr
Out[46]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [47]: arr.T
Out[47]: array([[1, 4, 7],
                [2, 5, 1],
                [3, 6, 0]])
```

- **Flat attribute**

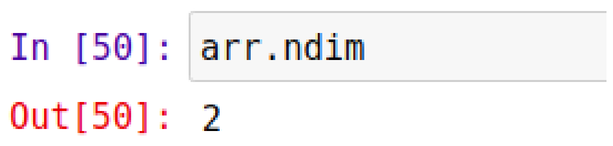
  This attribute gives the iterator and we can access the elements by iterating within a for loop.



```
In [48]: arr.flat
Out[48]: <numpy.flatiter at 0x2626370>

In [49]: for item in arr.flat:
             print(item)
         1
         2
         3
         4
         5
         6
         7
         1
         0
```

- **ndim attribute**

  'ndim' attribute gives the dimension of the array.



```
In [50]: arr.ndim
Out[50]: 2
```

  As we can see that the dimension of **'arr'** is 2 which represents that the array is of 2 dimensions.

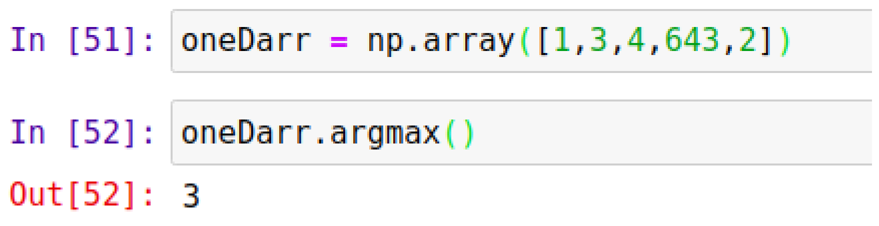
- **nbytes attribute**

  'nbytes' attribute provides the total bytes consumed by elements of an array.

=> Methods In One Dimensional Array

- **argmax( )**

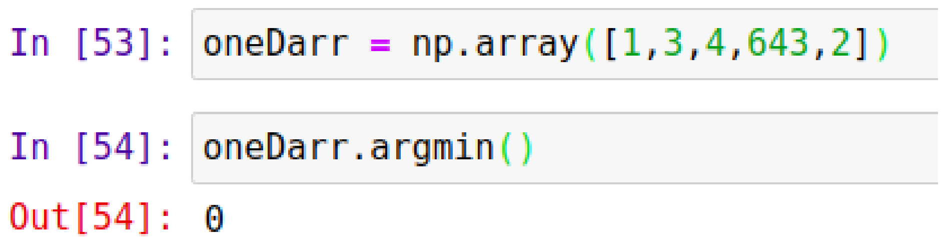
  This method provides the index of an array that has the maximum value.



```
In [51]: oneDarr = np.array([1,3,4,643,2])

In [52]: oneDarr.argmax()
Out[52]: 3
```

- **argmin( )**

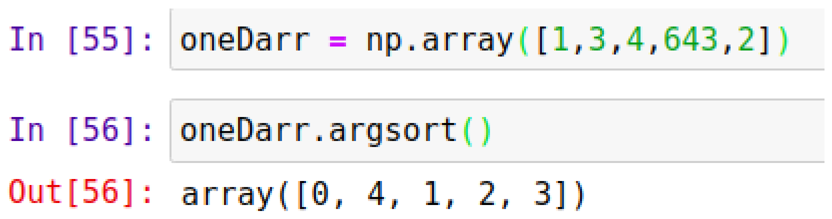
  This method provides the index of an array that has the minimum value.



```
In [53]: oneDarr = np.array([1,3,4,643,2])

In [54]: oneDarr.argmin()
Out[54]: 0
```

- **argsort()**

This method gives the index of an array in a sorted order.



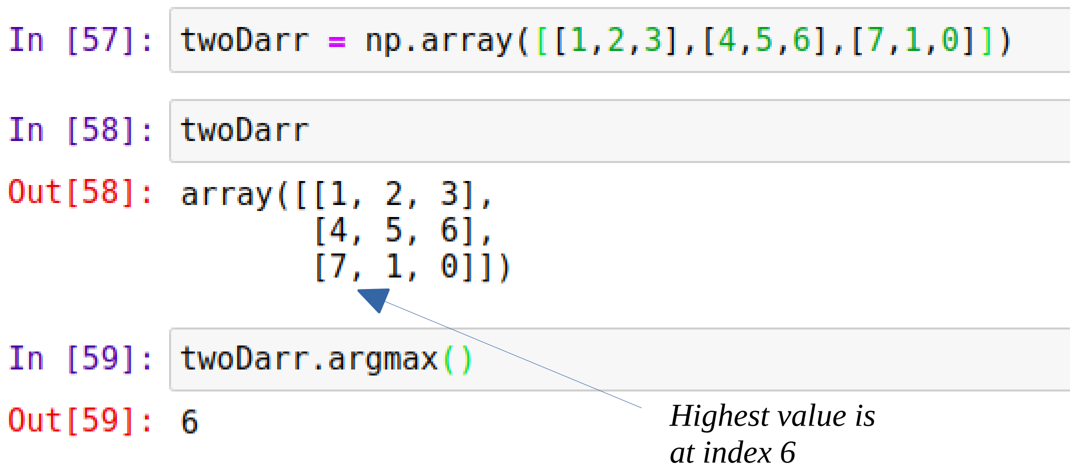
```
In [55]: oneDarr = np.array([1,3,4,643,2])

In [56]: oneDarr.argsort()
Out[56]: array([0, 4, 1, 2, 3])
```

Note: *It doesn't provide the sorted elements of an array but only provides the index of an array that needs to be in a given order, to get the elements of an array in a sorted manner.*

=> Methods In Two Dimensional Array

- **argmax( )**

In the case of two dimensional array if we apply the **'argmax( )'** method, at first it changes two dimensional array in a linear form and index are assigned to the elements in a continuous fashion starting with the index 0.



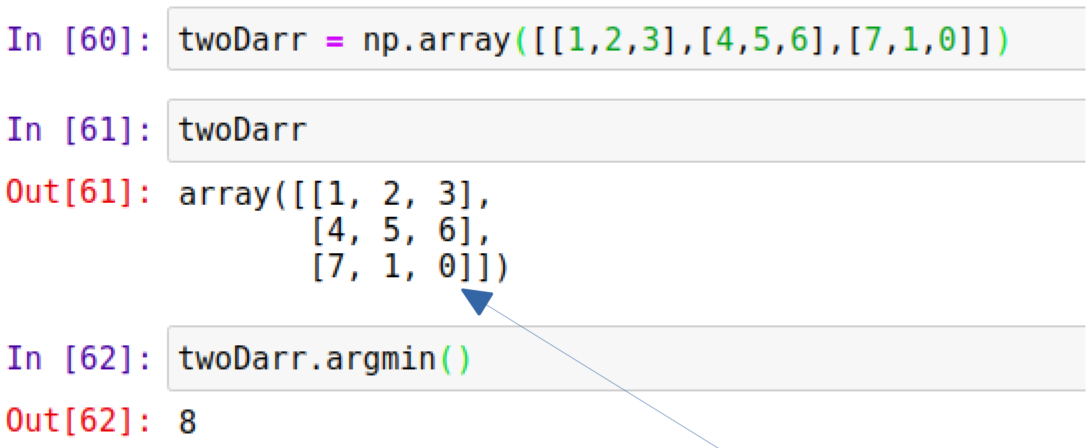
```
In [57]: twoDarr = np.array([[1,2,3],[4,5,6],[7,1,0]])

In [58]: twoDarr
Out[58]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [59]: twoDarr.argmax()
Out[59]: 6
```

Highest value is at index 6

After that **'argmax( )'** method provides the index of an array that has the highest value.

By Dipendra Shrestha

- **argmin( )**



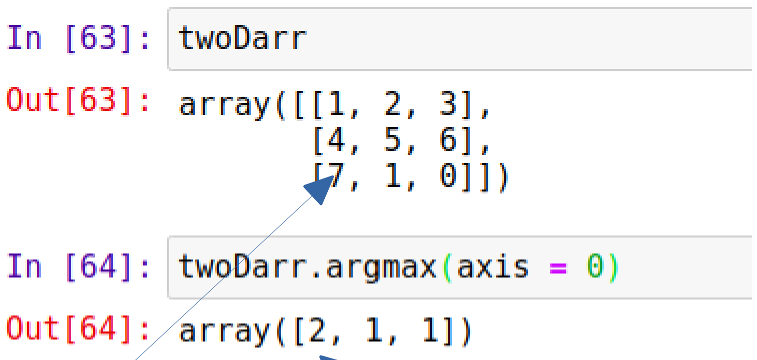
```
In [60]: twoDarr = np.array([[1,2,3],[4,5,6],[7,1,0]])

In [61]: twoDarr

Out[61]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [62]: twoDarr.argmin()

Out[62]: 8
```

Lowest value is at index 8

NOTE *: In order to get the* ***max*** *and* ***min*** *elements we can use* ***'.min( )'*** *and* ***'.max( )'*** *function.*

- **argmax (axis = 0)**

At first '**argmax '** finds the largest elements within a each row and gives the index of each row that has the largest value.

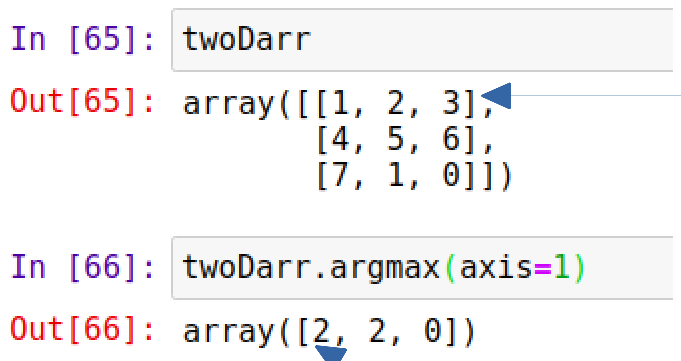


```
In [63]: twoDarr

Out[63]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [64]: twoDarr.argmax(axis = 0)

Out[64]: array([2, 1, 1])
```

Index of 1=> 0
Index of 4=> 1
Index of 7=> 2
The maximum value is at index '2'
and the result it produces is '2'

- **argmax(axis = 1)**

At first '**argmax** ' finds the largest elements within a each column and gives the index of each row that has the largest value.

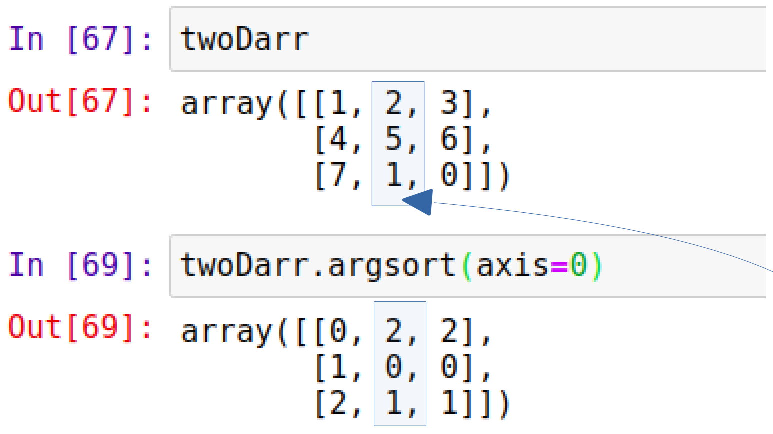


```
In [65]: twoDarr
Out[65]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [66]: twoDarr.argmax(axis=1)
Out[66]: array([2, 2, 0])
```

Index of 1=> 0
Index of 2=> 1
Index of 2=> 2
The maximum value is at index '2' and the result it produces is '2'

- **argsort(axis = 0)**

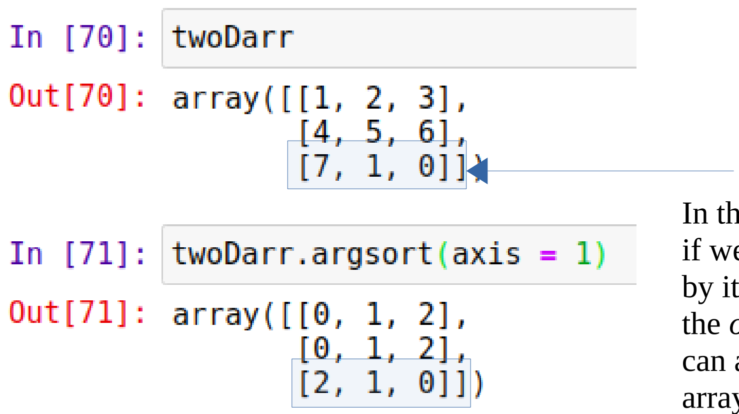
This method gives row wise index that needs to be arranged in order to sort the elements within a each row.



```
In [67]: twoDarr
Out[67]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [69]: twoDarr.argsort(axis=0)
Out[69]: array([[0, 2, 2],
                [1, 0, 0],
                [2, 1, 1]])
```

In the 2nd column of a matrix if we arrange the elements by its index as given in the *output section* then we can achieve the sorted array. Same case applies to 3rd column

- **argsort(axis = 1)**

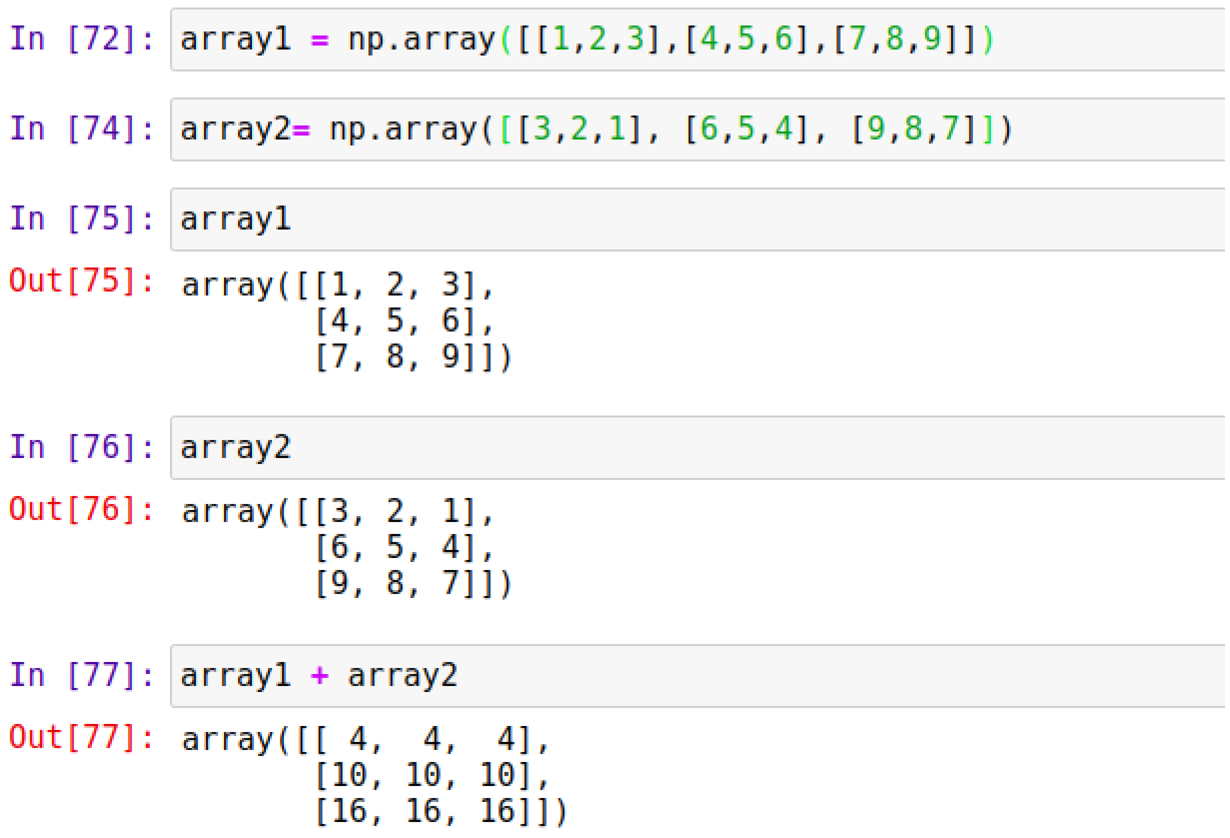


```
In [70]: twoDarr

Out[70]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [71]: twoDarr.argsort(axis = 1)

Out[71]: array([[0, 1, 2],
                [0, 1, 2],
                [2, 1, 0]])
```

In the 3rd row of a matrix if we arrange the elements by its index as given in the *output section* then we can achieve the sorted array.

Matrix Operation in numPy

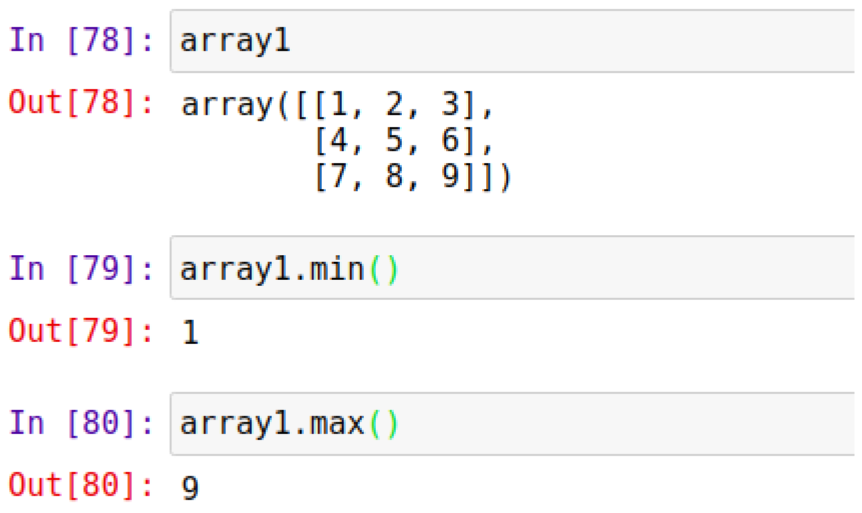
- **Addition of two matrices**



```
In [72]: array1 = np.array([[1,2,3],[4,5,6],[7,8,9]])

In [74]: array2= np.array([[3,2,1], [6,5,4], [9,8,7]])

In [75]: array1

Out[75]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [76]: array2

Out[76]: array([[3, 2, 1],
                [6, 5, 4],
                [9, 8, 7]])

In [77]: array1 + array2

Out[77]: array([[ 4,  4,  4],
                [10, 10, 10],
                [16, 16, 16]])
```

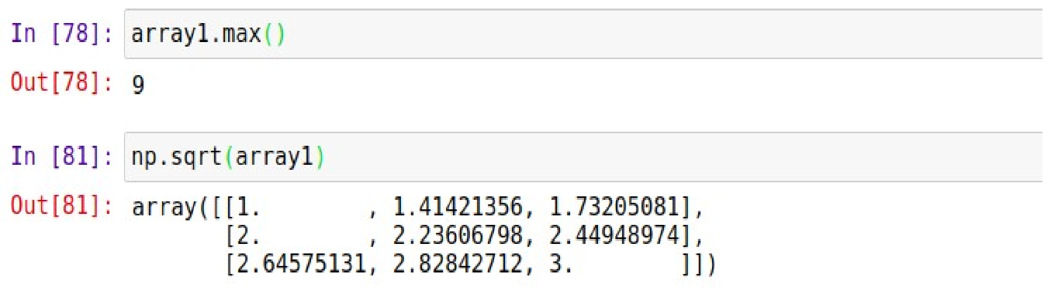
We can simply add two arrays using the '+' operator. Same like this we can perform ***subtraction, multiplication, division*** between two array.

To get the minimum and maximum elements within an array we can use ***'.min( )'*** and ***'max( )'*** methods.



```
In [78]: array1
Out[78]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [79]: array1.min()
Out[79]: 1

In [80]: array1.max()
Out[80]: 9
```

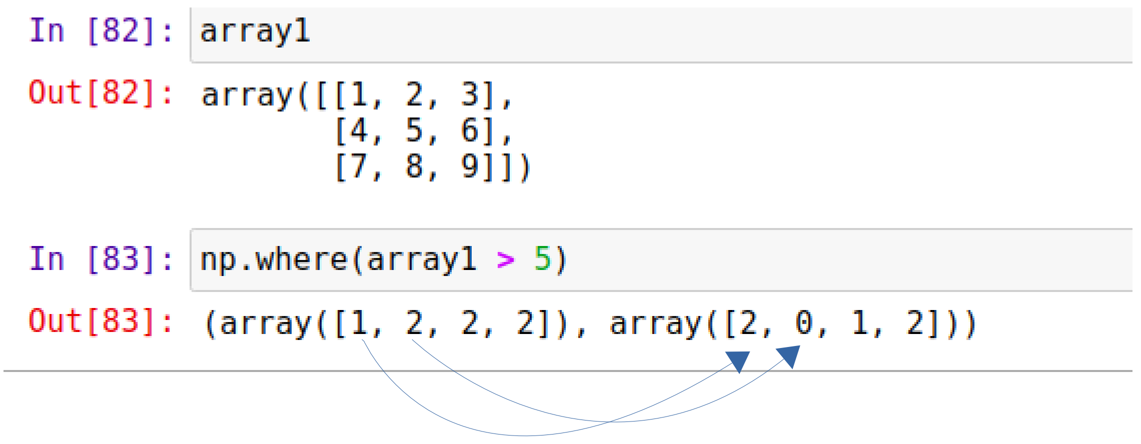
To get square root of the array elements we can use ***'.sqrt'*** **.**



```
In [78]: array1.max()
Out[78]: 9

In [81]: np.sqrt(array1)
Out[81]: array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974],
                [2.64575131, 2.82842712, 3.        ]])
```

Finding Elements within a numPy array.

To find the elements within a given array we can use **'where'** method as shown below.



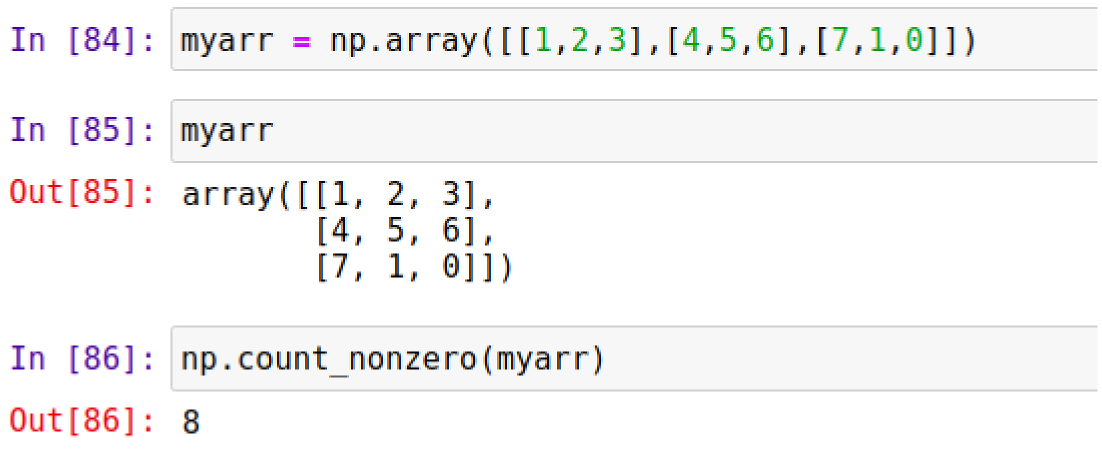
```
In [82]: array1
Out[82]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [83]: np.where(array1 > 5)
Out[83]: (array([1, 2, 2, 2]), array([2, 0, 1, 2]))
```

The index pair at which the value is greater than **5** are ***[1, 2], [2, 0], [2, 1], [2, 2].***

Non-zero method in numPy array

- **count_nonzero( )**

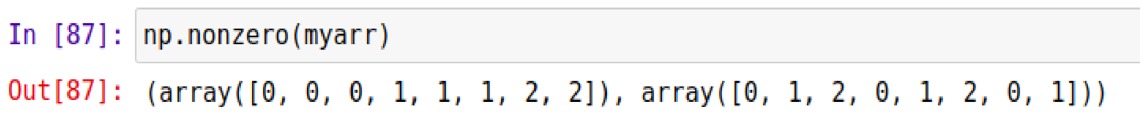
  This method provides the number of counts of zero elements present within an array.



```
In [84]: myarr = np.array([[1,2,3],[4,5,6],[7,1,0]])

In [85]: myarr

Out[85]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 1, 0]])

In [86]: np.count_nonzero(myarr)

Out[86]: 8
```

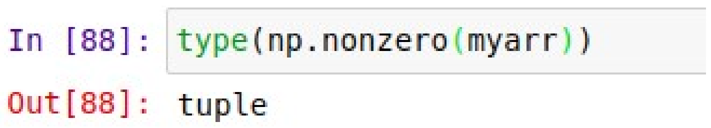
- **nonzero( )**

  This method provides the tuples for each axis , that has nonzero elements.



```
In [87]: np.nonzero(myarr)

Out[87]: (array([0, 0, 0, 1, 1, 1, 2, 2]), array([0, 1, 2, 0, 1, 2, 0, 1]))
```

In order to observe the datatype it yields, we can simply check the datatype by using **‘type( )’** method as shown below:

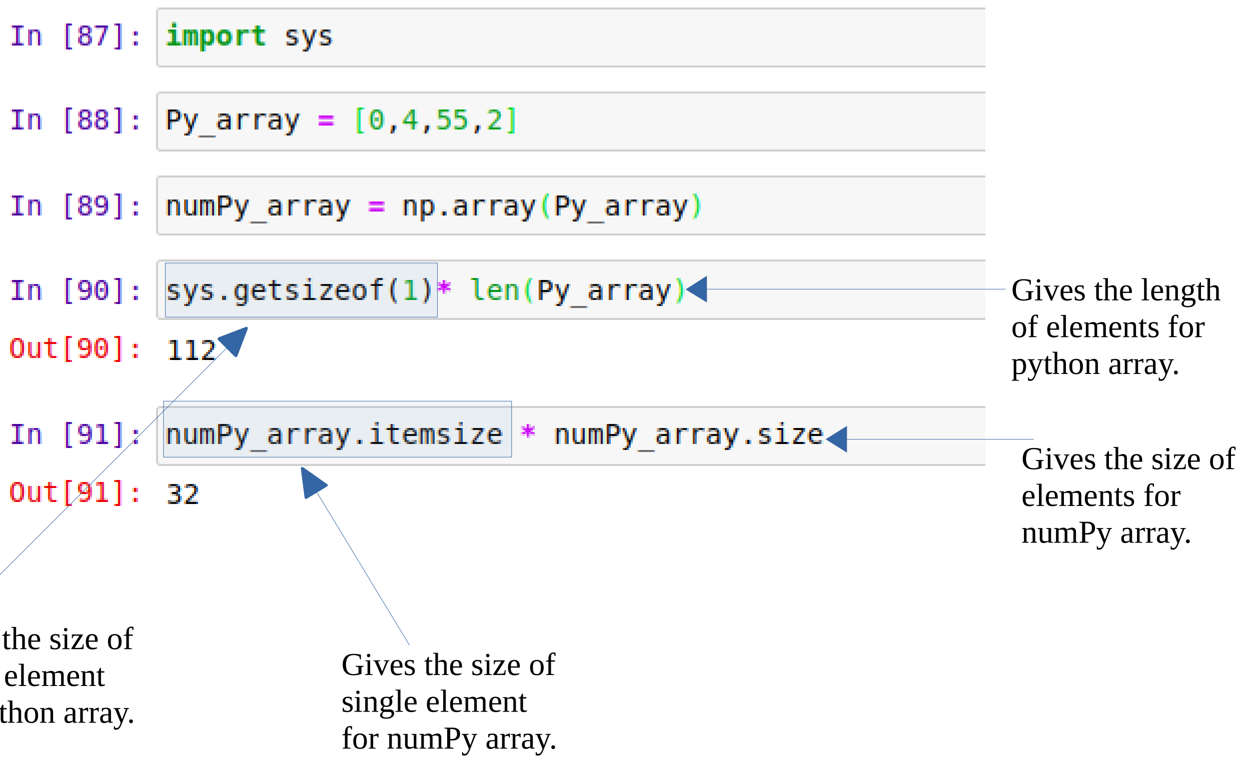


```
In [88]: type(np.nonzero(myarr))

Out[88]: tuple
```

By Dipendra Shrestha

Demonstration of How numPy Array Preserve the Memory Space

As shown in below example two arrays are created. One array is a python array whereas another array is of numPy array.
If we compare the size consumption by the elements between the python array and numPy array then we can see that the numPy array consumes very less memory.



```
In [87]: import sys

In [88]: Py_array = [0,4,55,2]

In [89]: numPy_array = np.array(Py_array)

In [90]: sys.getsizeof(1)* len(Py_array)
Out[90]: 112

In [91]: numPy_array.itemsize * numPy_array.size
Out[91]: 32
```

Gives the length of elements for python array.
Gives the size of elements for numPy array.
Gives the size of single element for Python array.
Gives the size of single element for numPy array.

REFERENCES

https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html