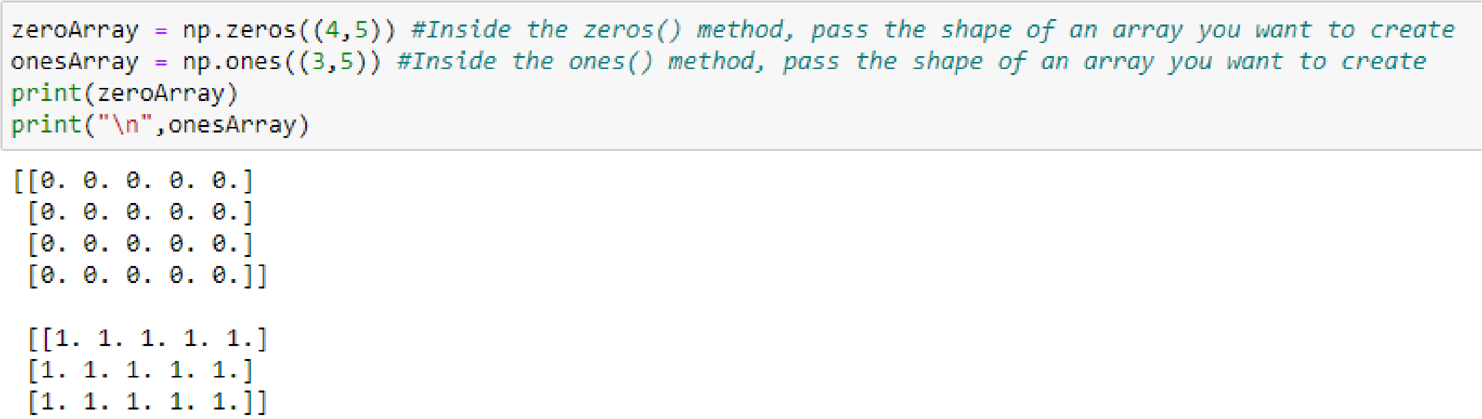
NUMPY

- It’s a **python library** , which stands for **“Numerical Python”,** used for working with **arrays**.
- It also has functions for working in the domain of **linear algebra, fourier transform, & matrices.**
- Numpy is similar to python list, but of the same data type which makes it faster compared to python list
- Numpy provides an array object called **ndarray[**n-dimensional array**]**.
- Numpy arrays are stored at **one continuous place** in memory unlike lists, so processes can access and manipulate them very efficiently.

- **Creating arrays :**
    - We can create a NumPy **ndarray** object by using **array()** function**.**
    - To create an **ndarray**, we can pass a list, tuple or array-like object into the **array()** method and it will be created into an **ndarray.**
    - Total number of **indices required to access an element** inside an array is known as the **dimension of that array.**
    - **Dimensions** in array:
        - Depth level of an array [Nested arrays]
        - **0-D** array : 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.
        - **1-D** array : An array that has **0-D arrays as its elements** is called a uni-dimensional or 1-D array.
        - **2-D** array : An array that has **1-D arrays as its elements** is called a 2-D array. These are often used to represent **matrix or 2nd order tensors**.
        - **3-D** array : An array that has **2-D arrays (matrices) as its elements** is called 3-D array. These are often used to represent a 3rd order tensor.
    - NumPy Arrays provides the **ndim** attribute that returns an integer that tells us how many dimensions the array has.
    - Also we can create higher dimensional arrays using the **ndmin** attribute. [**<**arrayName**>.ndim**]
    - To get the item size of an array, we use **itemsize** property. [**<**arrrayName**>.itemsize**]
    - To get the size of an array[No. of elements], we use **size** property. [**<**arrayName**>.size**]
    - If we want to create an array with all the elements filled with 0, then we use the **zeros()** method instead of **array().**

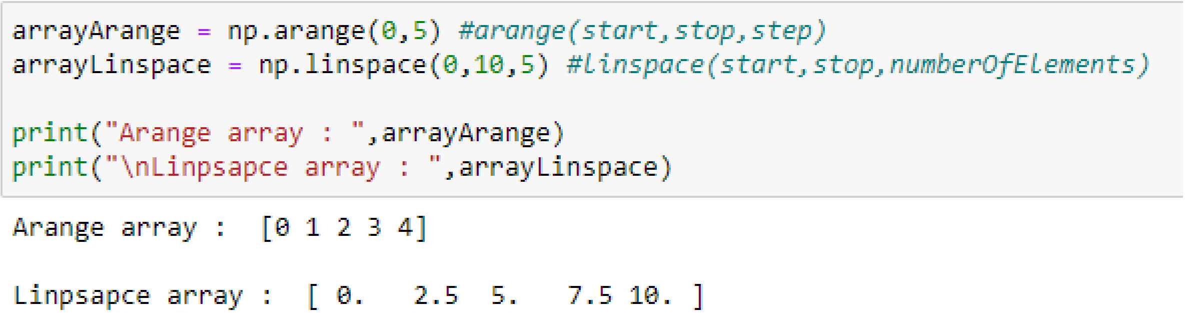
- If we want to create an array with all the elements filled with 1, then we use the **ones()** method instead of **array().**



```
zeroArray = np.zeros((4,5)) #Inside the zeros() method, pass the shape of an array you want to create
onesArray = np.ones((3,5)) #Inside the ones() method, pass the shape of an array you want to create
print(zeroArray)
print("\n",onesArray)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

 [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

- We have a function similar to **range(),** i.e., **arange()** which does the same operation. [**arange(**<start>**,**<stop>**,**<step>**)**]. **Stop** is not included.
- If we want to create an array with all the elements linearly spaced, we use **linspace()** . [**linspace(**<start>**,**<stop>**,**<numberOfElements>**)**]



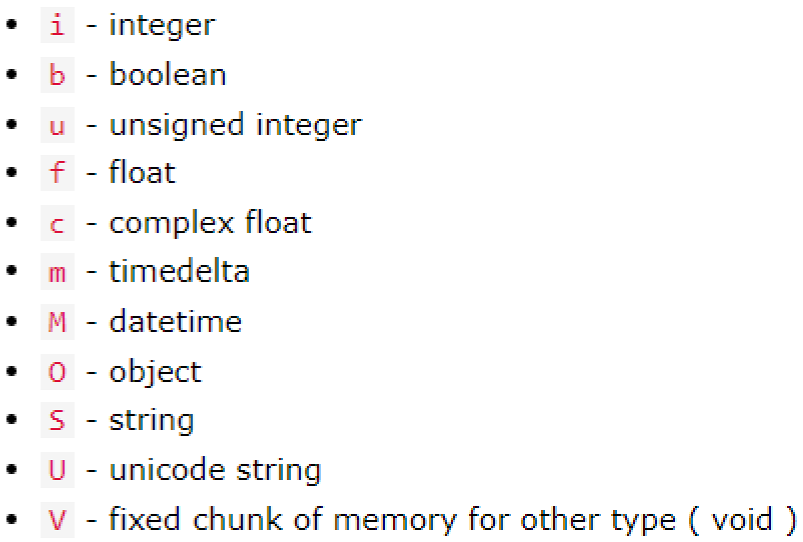
```
arrayArange = np.arange(0,5) #arange(start,stop,step)
arrayLinspace = np.linspace(0,10,5) #linspace(start,stop,numberOfElements)

print("Arange array : ",arrayArange)
print("\nLinpsapce array : ",arrayLinspace)
```

```
Arange array :  [0 1 2 3 4]

Linpsapce array :  [ 0.   2.5  5.   7.5 10. ]
```

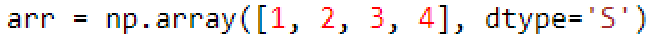
- **NumPy Array Indexing:**
  - Array indexing is the same as accessing an array element.
  - Accessing 2D array : using two integers representing rows and columns.
  - Accessing 3D array : using 3 integers representing the rows, columns and index of the element.
  - Negative indexing : used to access the elements from the end of an array.

- **NumPy Array Slicing:**
  - Slicing is used to get elements from one index value till another index value
  - **[**<start>**,**<stop>**,**<step>**]**
    - **start :** starting index value. If not passed, will be considered as **0**
    - **stop :** ending index value. If not passed, it will be considered till the end of the array. If passed, that will be excluded.
    - **step :** step size. Default will be **1.**

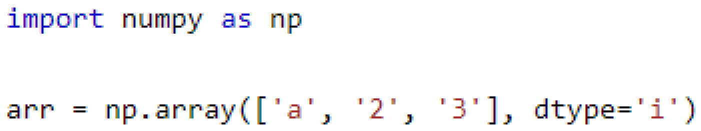
- **NumPy Data Types:**



- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

  - **dtype** returns the data type of the **ndarray.**
  - We can create arrays with different data types by providing **dtype** values while creating arrays.
  - 

```
arr = np.array([1, 2, 3, 4], dtype='S')
```

  - If a type is given in which elements can't be casted then NumPy will raise a **ValueError**.
  - 

```
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```

  - **Converting data type on existing arrays :**
    - Best way to change the data type of an array is to make a copy of that array using **astype()** method.
    - The **astype()** method creates a copy of the array and allows you to specify the data type as a parameter.
    - You can specify the data type as a string like ‘f’ for float, ‘i’ for integer or you can use the data type directly like float, integer etc

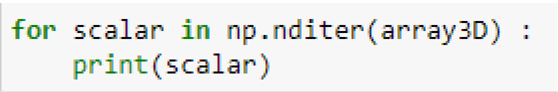
- **NumPy Copy vs View:**
  - **copy()** is the new array and owns the data. Changes made to the copy array will not affect the original array, and vice-versa is also true.
  - **view()** is just the view of an array. It doesn’t own any data and changes made to the original array affects the view and vice-versa holds too.

  - Check if array owns its data :
    - **base** attribute is used to check if the array owns its data.
    - If it returns **none**, then that array **owns** the data.
    - If it returns any array, then it doesn’t own any data. Returned array is its base array.

- **NumPy Array Shape:**
  - Shape of an array is the number of elements in each dimension.
  - **shape** attribute returns the tuple with each index having the number of corresponding elements.

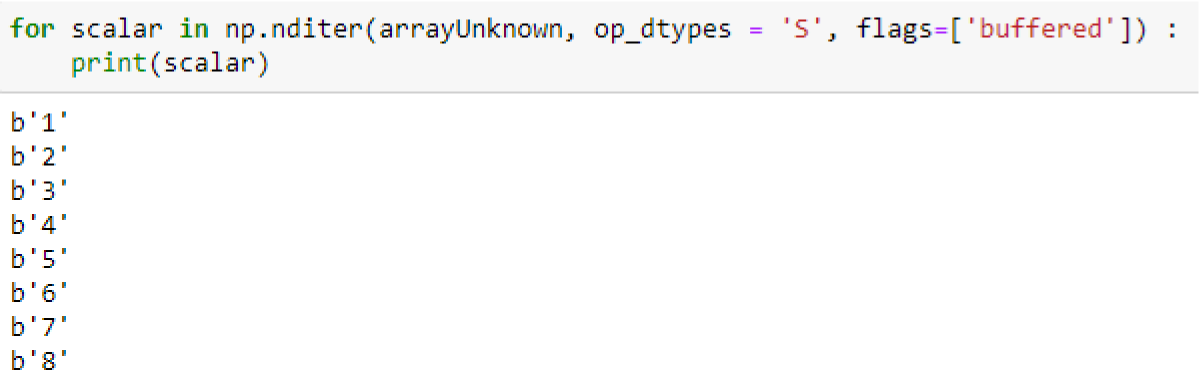
- **NumPy Reshaping:**
  - Reshaping means changing the shape of an array.
  - By adding or removing dimensions or changing the number of elements in each dimension.
  - **reshape()** method is used.
  - As long as the elements required for reshaping are equal in both shapes, we can reshape for any dimension.
  - The new reshaped array should be stored in another array for it to make it genuine. Or else it’ll return as a **view**.
  - You are allowed to have one "unknown" dimension. Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method. Pass **-1** as the value, and NumPy will calculate this number for you. But the check in here is that you cannot pass **-1** to more than one dimension.
  - **Flattening an array** means reshaping any dimension array into a 1D array**. reshape(-1)** to do the task. Or we can use the **ravel()** method without passing any inputs.

- **NumPy Array Iterating :**
  - Using **for** loop to iterate through each value, we need **n** loops to iterate **n**-D array.
  - Using **nditer()** method, we can iterate any dimension array in **one for loop**.



```
for scalar in np.nditer(array3D) :
    print(scalar)
```

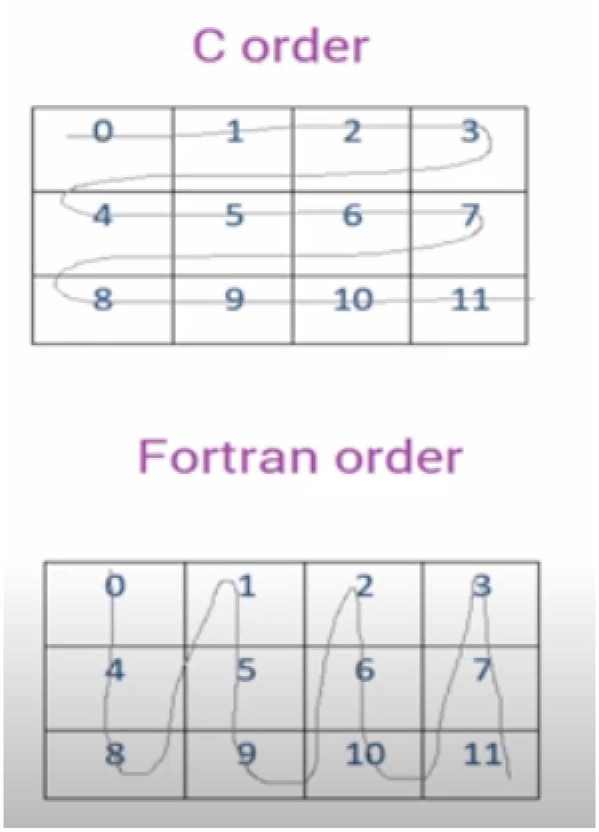
  - We can use the **op_dtypes** argument and pass it the expected datatype to change the datatype of elements while iterating. NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in **nditer()** we pass **flags=['buffered']**.



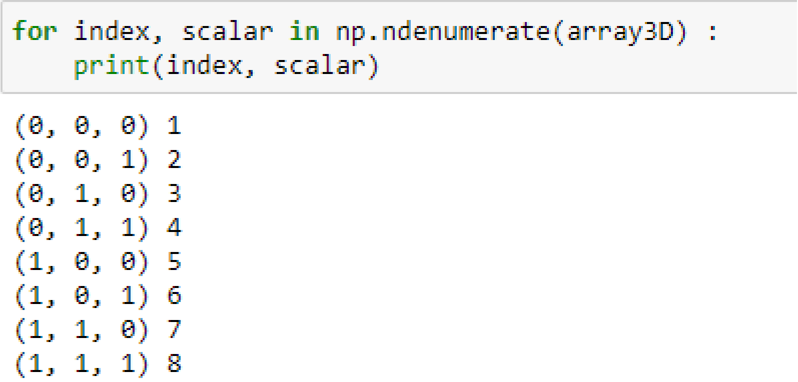
```
for scalar in np.nditer(arrayUnknown, op_dtypes = 'S', flags=['buffered']) :
    print(scalar)
```

```
b'1'
b'2'
b'3'
b'4'
b'5'
b'6'
b'7'
b'8'
```

- We can pass the **order** by which the control must flow:
    - **C** : goes through row by row
    - **F :** goes through column



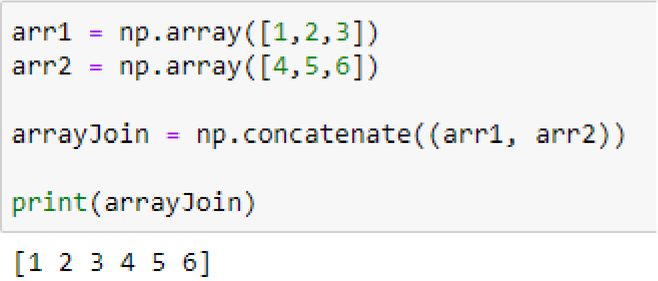
- Enumerated Iteration Using **ndenumerate():**
    - Mentioning sequence number of something one by one.
    - When we require the index number of elements, we can use the above method to get that



```
for index, scalar in np.ndenumerate(array3D) :
    print(index, scalar)
```

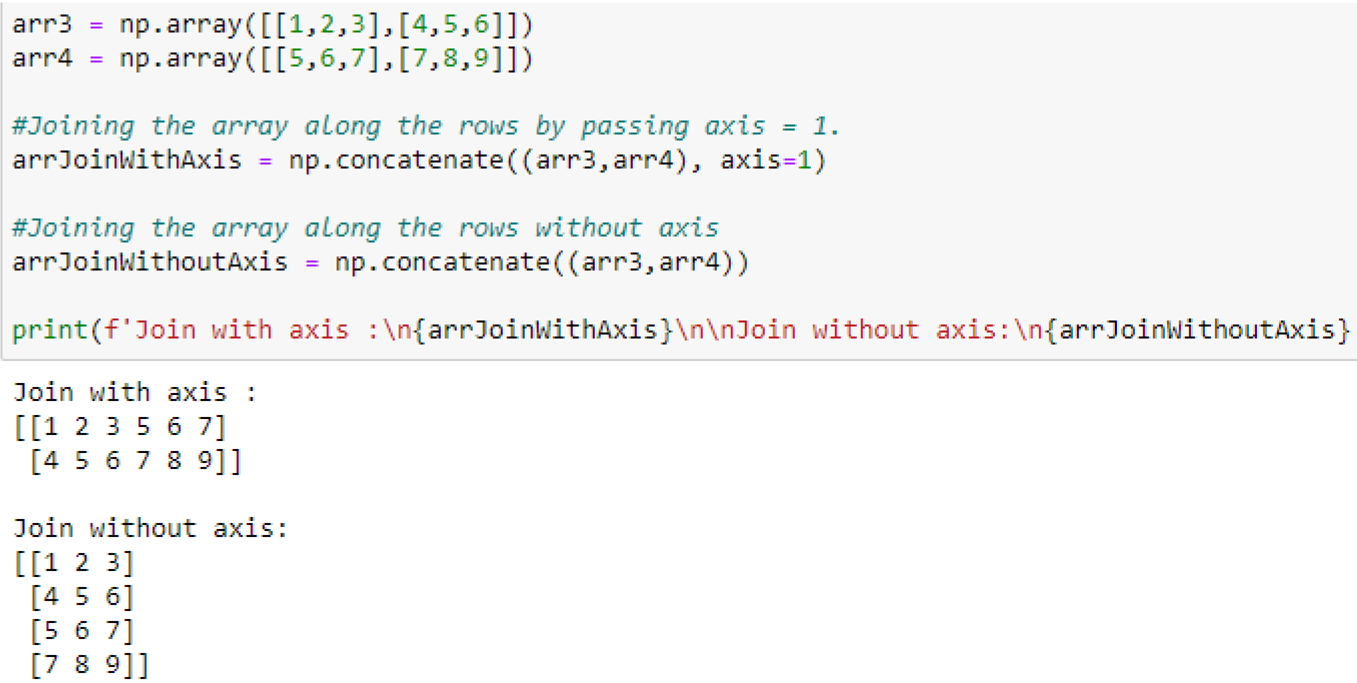
```
(0, 0, 0) 1
(0, 0, 1) 2
(0, 1, 0) 3
(0, 1, 1) 4
(1, 0, 0) 5
(1, 0, 1) 6
(1, 1, 0) 7
(1, 1, 1) 8
```

- **NumPy Array Join :**
  - Join means putting contents of two or more arrays into one array.
  - In SQL, we join tables based on **key**, but in NumPy we join arrays based on **axis**.
  - **concatenate()** method is used to join arrays.



```
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])

arrayJoin = np.concatenate((arr1, arr2))

print(arrayJoin)
```

```
[1 2 3 4 5 6]
```

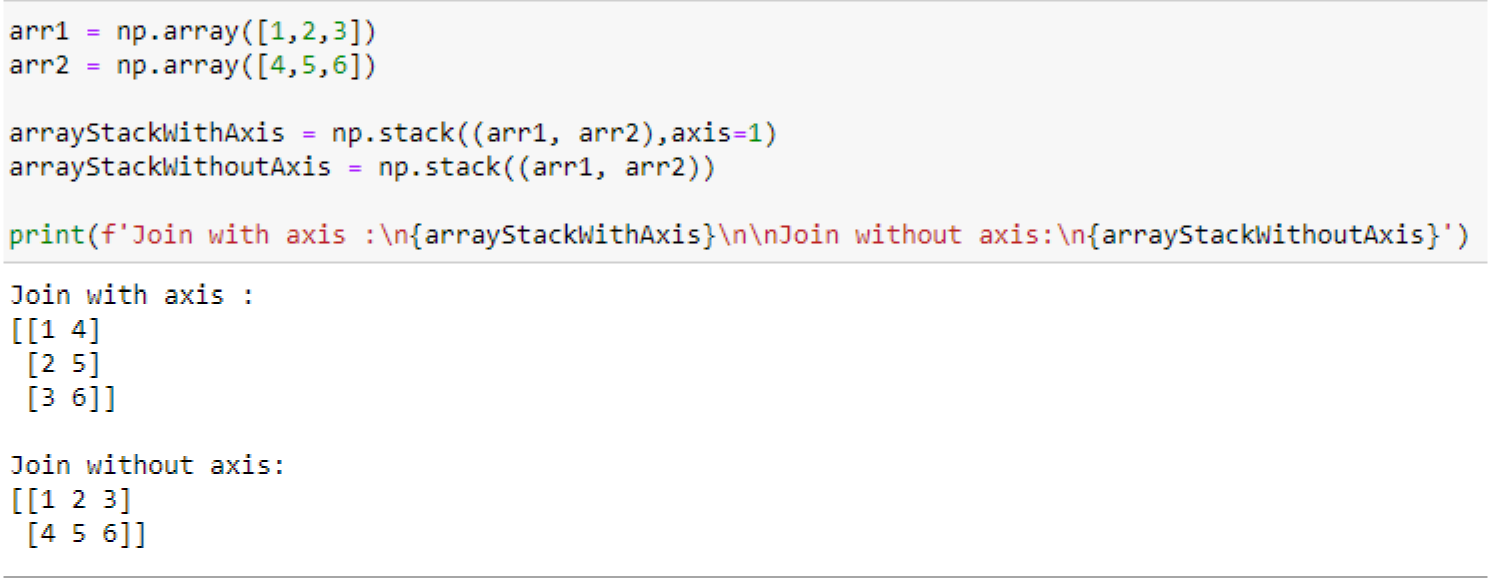


```
arr3 = np.array([[1,2,3],[4,5,6]])
arr4 = np.array([[5,6,7],[7,8,9]])

#Joining the array along the rows by passing axis = 1.
arrJoinWithAxis = np.concatenate((arr3,arr4), axis=1)

#Joining the array along the rows without axis
arrJoinWithoutAxis = np.concatenate((arr3,arr4))

print(f'Join with axis :\n{arrJoinWithAxis}\n\nJoin without axis:\n{arrJoinWithoutAxis}
```

```
Join with axis :
[[1 2 3 5 6 7]
 [4 5 6 7 8 9]]

Join without axis:
[[1 2 3]
 [4 5 6]
 [5 6 7]
 [7 8 9]]
```

  - You have to pass an **axis**, otherwise it joins separately.
  - **Joining Arrays Using Stack Functions** :
    - Same as concatenation, but stacking is done along a new axis.

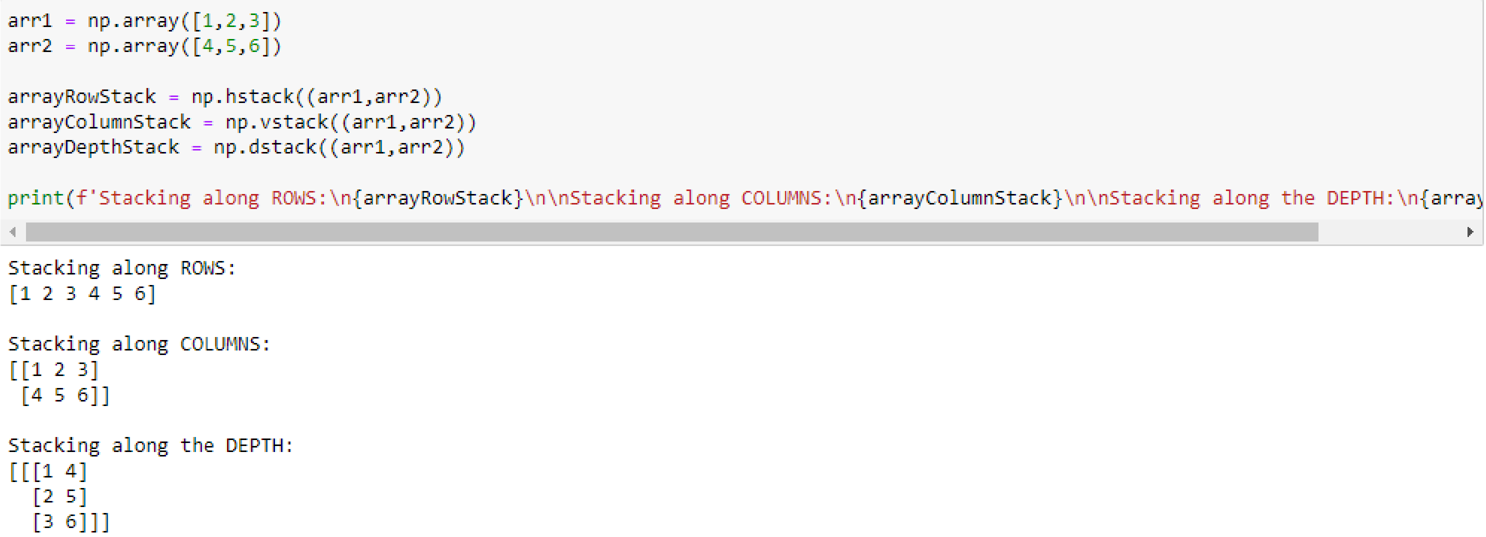
    - We pass a sequence of arrays that we want to join to the **stack()** method along with the axis. If the axis is not explicitly passed it is taken as 0.



```
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])

arrayStackWithAxis = np.stack((arr1, arr2),axis=1)
arrayStackWithoutAxis = np.stack((arr1, arr2))

print(f'Join with axis :\n{arrayStackWithAxis}\n\nJoin without axis:\n{arrayStackWithoutAxis}')
```

```
Join with axis :
[[1 4]
 [2 5]
 [3 6]]

Join without axis:
[[1 2 3]
 [4 5 6]]
```

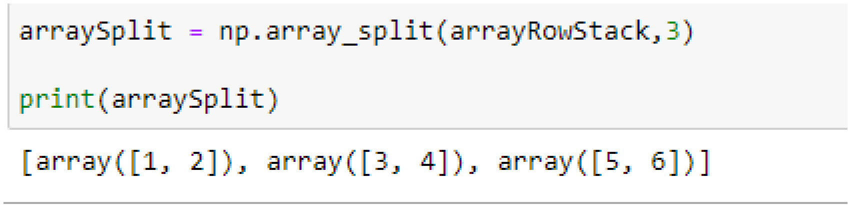
    - **hstack() :** to stack along **rows**
    - **vstack() :** to stack along **columns**
    - **dstack() :** to stack along **depth or height**



```
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])

arrayRowStack = np.hstack((arr1,arr2))
arrayColumnStack = np.vstack((arr1,arr2))
arrayDepthStack = np.dstack((arr1,arr2))

print(f'Stacking along ROWS:\n{arrayRowStack}\n\nStacking along COLUMNS:\n{arrayColumnStack}\n\nStacking along the DEPTH:\n{array
```

```
Stacking along ROWS:
[1 2 3 4 5 6]

Stacking along COLUMNS:
[[1 2 3]
 [4 5 6]]

Stacking along the DEPTH:
[[[1 4]
  [2 5]
  [3 6]]]
```

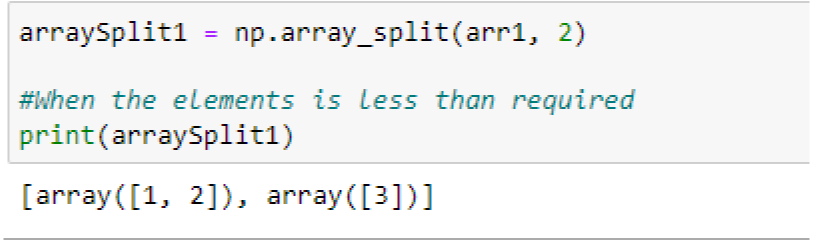
- **NumPy Splitting Array :**
  - Reverse operation of joining.
  - Breaks a single array into multiple arrays.
  - **array_split(),** we pass the array which we want to split and the number of splits we want.



```
arraySplit = np.array_split(arrayRowStack,3)

print(arraySplit)
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

- The return value will be a **list** containing the splitted arrays.
- If the array has less number of elements than required, then it will adjust from end.

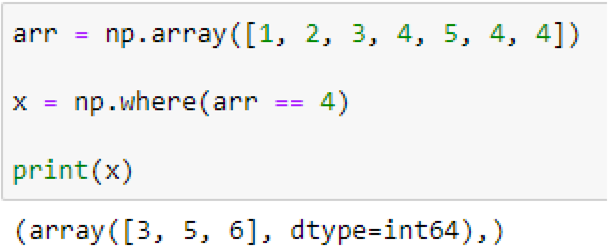


```
arraySplit1 = np.array_split(arr1, 2)

#When the elements is less than required
print(arraySplit1)
```

```
[array([1, 2]), array([3])]
```

- Use the **hsplit()** method to split the 2-D array into three 2-D arrays along **rows**.
- Similarly we have **vsplit(), dsplit().**

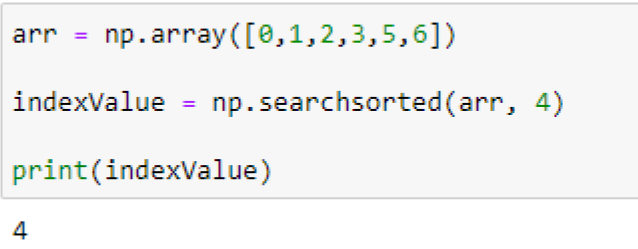
- **NumPy Array Search :**
  - You can search for a specific value in the array and return their index value.
  - **where()** method is used for that.



```
arr = np.array([1, 2, 3, 4, 5, 4, 4])

x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6], dtype=int64),)
```

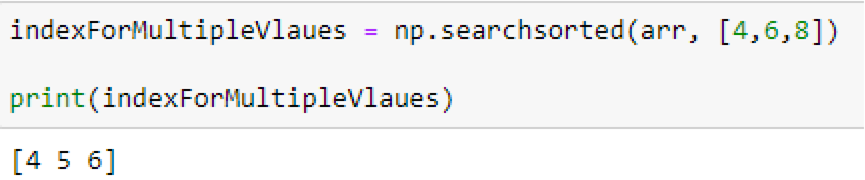
  - This returns a **tuple** which contains all the index values where the specified value matches inside the array.
  - **searchsorted()** : performs a binary search in the array and returns the index where specified value would be inserted to maintain the search order.



```
arr = np.array([0,1,2,3,5,6])

indexValue = np.searchsorted(arr, 4)

print(indexValue)
```

```
4
```

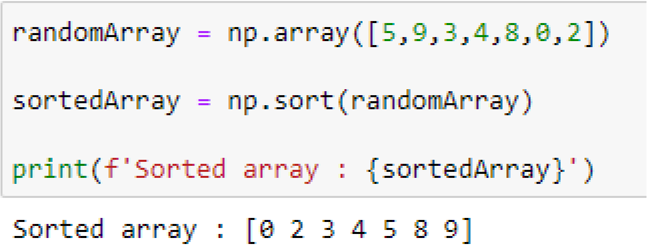
  - By default the search starts from the left**.** If we want the search to start from right, then pass **side = ‘right’** while doing the operation.
  - To search more than one value, use an array with the specified values.



```
indexForMultipleVlaues = np.searchsorted(arr, [4,6,8])

print(indexForMultipleVlaues)
```

```
[4 5 6]
```

- **NumPy Sorting Arrays** :
  - Sorting means putting elements in an ordered sequence.
  - **sort() :** method used to sort the given array.

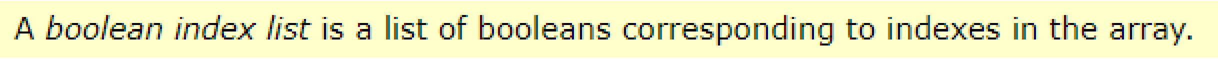


```
randomArray = np.array([5,9,3,4,8,0,2])

sortedArray = np.sort(randomArray)

print(f'Sorted array : {sortedArray}')
```

```
Sorted array : [0 2 3 4 5 8 9]
```

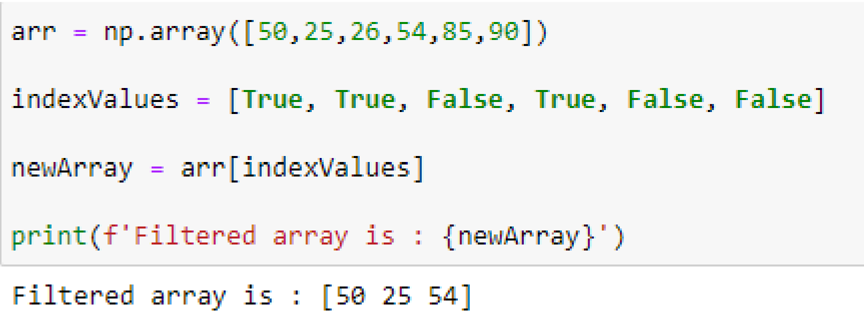
  - The type doesn't matter for sorting. It might be numerical(ascending or descending), alphabetical order.

- **NumPy Filter Array :**
  - Getting some elements from an existing array and creating a new array out of them is called filtering.
  - In NumPy, you filter an array using a boolean index list.



A *boolean index list* is a list of booleans corresponding to indexes in the array.

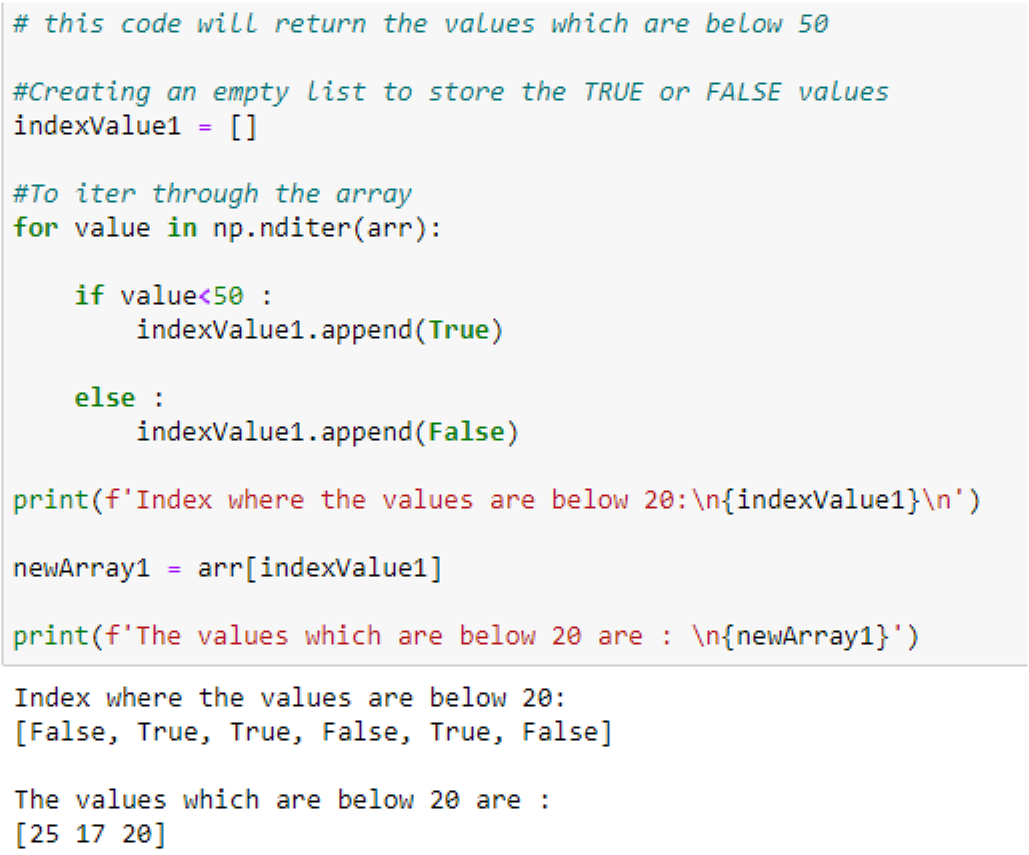
  - If the value at an index is **True** that element is **contained** in the filtered array, if the value at that index is **False** that element is **excluded** from the filtered array.



```
arr = np.array([50,25,26,54,85,90])

indexValues = [True, True, False, True, False, False]

newArray = arr[indexValues]

print(f'Filtered array is : {newArray}')
```

```
Filtered array is : [50 25 54]
```

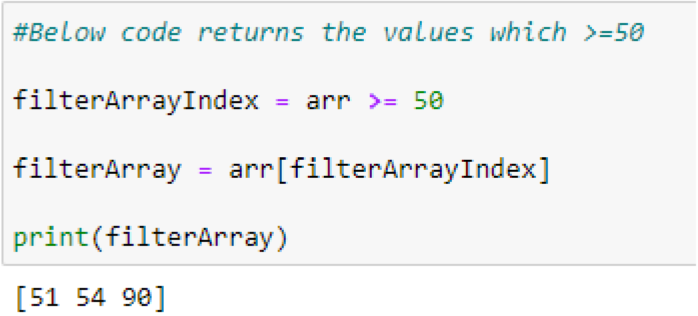
  - The above one is hard coded, but the common practice is condition based.



```
# this code will return the values which are below 50

#Creating an empty list to store the TRUE or FALSE values
indexValue1 = []

#To iter through the array
for value in np.nditer(arr):

    if value<50 :
        indexValue1.append(True)

    else :
        indexValue1.append(False)

print(f'Index where the values are below 20:\n{indexValue1}\n')

newArray1 = arr[indexValue1]

print(f'The values which are below 20 are : \n{newArray1}')
```

```
Index where the values are below 20:
[False, True, True, False, True, False]

The values which are below 20 are :
[25 17 20]
```

- ○ We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.



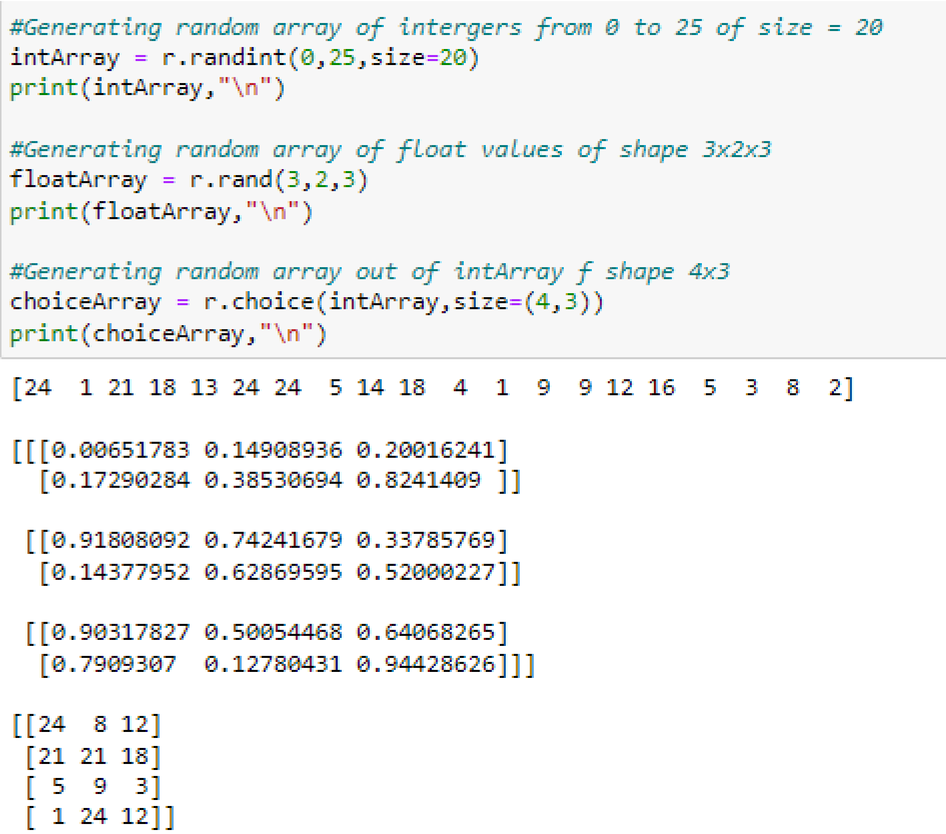
```
#Below code returns the values which >=50

filterArrayIndex = arr >= 50

filterArray = arr[filterArrayIndex]

print(filterArray)
```

```
[51 54 90]
```

- **Basic Mathematical Functions:**
  - **min() :** gives the minimum value of entire array
  - **max() :** gives the maximum value of entire array
  - **sum() :** gives the addition value of entire array
  - **sqrt() :** gives the square root each element of entire array
  - **std() :** gives the standard deviation of entire array

NumPy Random :

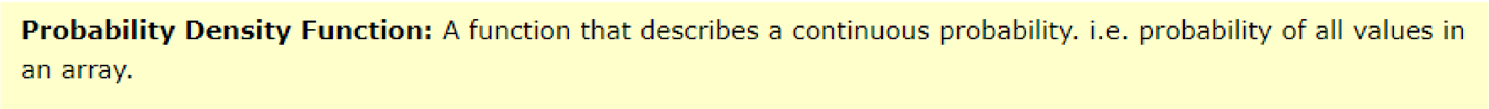
- Random numbers do not mean a different number every time. Random means something you can't predict logically.
- **random** module from python offers to work with random numbers.
- **randint(<**value**>) :** to generate random integer within the <**value**> specified.
- **rand(<**shape**>) :** to generate random float numbers between **0 & 1.** Returns a random **array** in the specified shape.
- By passing the **size** parameter to **randint()**, we can generate a random array of specified shape and size.
- **choice() :** method takes an array as parameter and randomly returns a value out of array values.
- By passing **size** to **choice()** method, it returns an array of values of that size and shape.



```
#Generating random array of intergers from 0 to 25 of size = 20
intArray = r.randint(0,25,size=20)
print(intArray,"\n")

#Generating random array of float values of shape 3x2x3
floatArray = r.rand(3,2,3)
print(floatArray,"\n")

#Generating random array out of intArray f shape 4x3
choiceArray = r.choice(intArray,size=(4,3))
print(choiceArray,"\n")
```

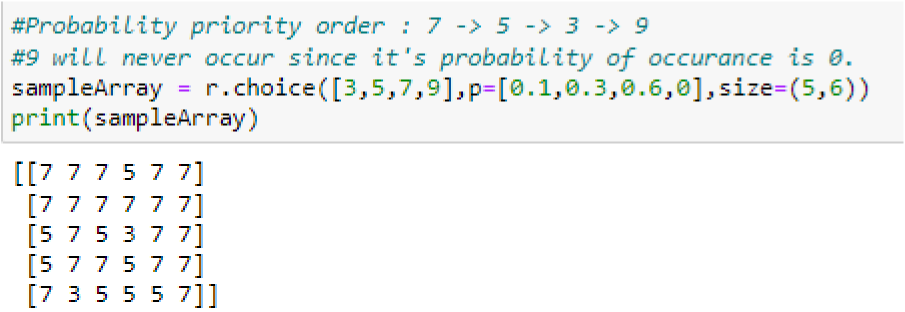
```
[24  1 21 18 13 24 24  5 14 18  4  1  9  9 12 16  5  3  8  2]

[[[0.00651783 0.14908936 0.20016241]
  [0.17290284 0.38530694 0.8241409 ]]

 [[0.91808092 0.74241679 0.33785769]
  [0.14377952 0.62869595 0.52000227]]

 [[0.90317827 0.50054468 0.64068265]
  [0.7909307  0.12780431 0.94428626]]]

[[24  8 12]
 [21 21 18]
 [ 5  9  3]
 [ 1 24 12]]
```

- **Random Data Distribution :**
  - **Data distribution** is the list of all possible values and how often each value occurs.
  - **Random distribution** is a set of random numbers that follow a certain **probability density function.**



Probability Density Function: A function that describes a continuous probability. i.e. probability of all values in an array.

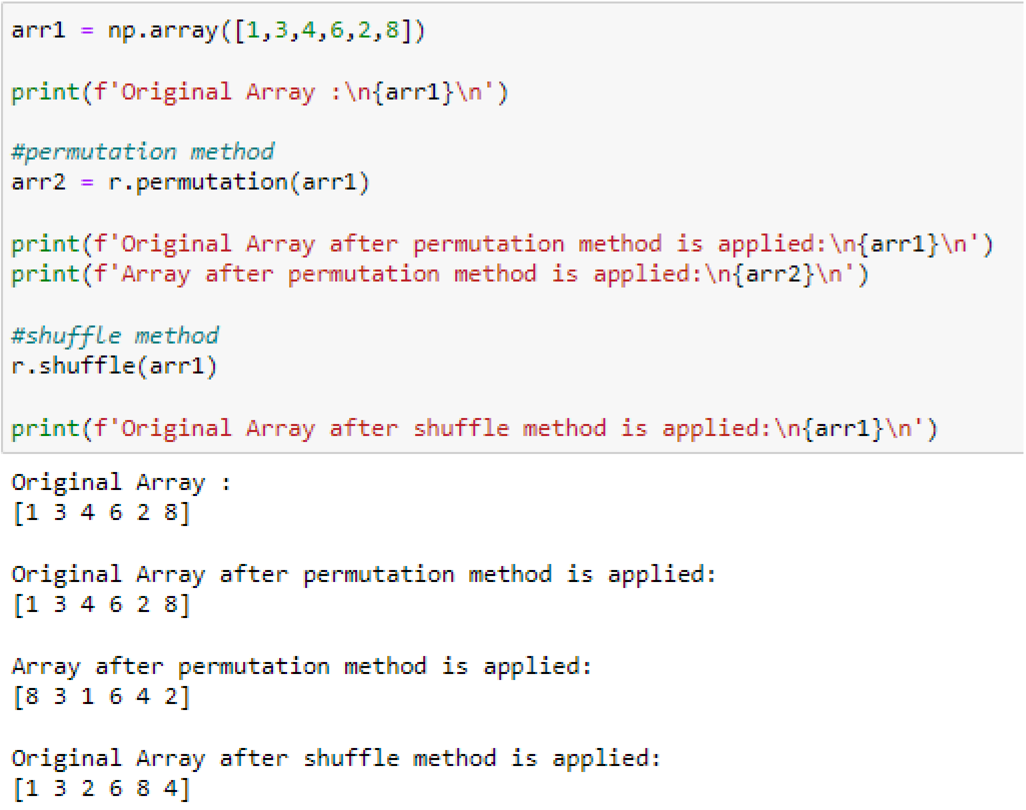
- Probability is set by a number between 0 & 1. Where **0** means value will **never occur** and **1** means value will **always occur**.
- Sum of all probabilities should be equal to 1.



```
#Probability priority order : 7 -> 5 -> 3 -> 9
#9 will never occur since it's probability of occurance is 0.
sampleArray = r.choice([3,5,7,9],p=[0.1,0.3,0.6,0],size=(5,6))
print(sampleArray)
```

```
[[7 7 7 5 7 7]
 [7 7 7 7 7 7]
 [5 7 5 3 7 7]
 [5 7 7 5 7 7]
 [7 3 5 5 5 7]]
```

- **Random Permutation :**
  - Permutation refers to the arrangement of elements in different ways.
  - We have two methods for this :
    - **shuffle():** shuffling means changing arrangement of elements in-place i.e., within an array. It makes changes to the original array.
    - **permutation():** this method returns the re-arranged array and leaves the original array un-touched.

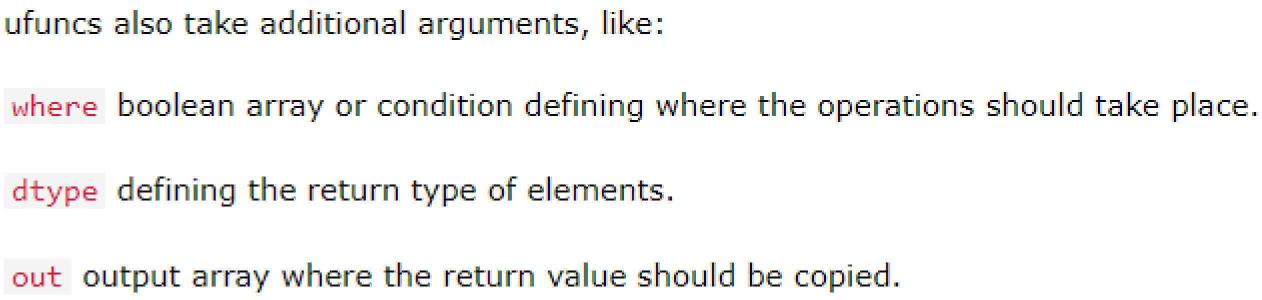


```
arr1 = np.array([1,3,4,6,2,8])

print(f'Original Array :\n{arr1}\n')

#permutation method
arr2 = r.permutation(arr1)

print(f'Original Array after permutation method is applied:\n{arr1}\n')
print(f'Array after permutation method is applied:\n{arr2}\n')

#shuffle method
r.shuffle(arr1)

print(f'Original Array after shuffle method is applied:\n{arr1}\n')
```

```
Original Array :
[1 3 4 6 2 8]

Original Array after permutation method is applied:
[1 3 4 6 2 8]

Array after permutation method is applied:
[8 3 1 6 4 2]

Original Array after shuffle method is applied:
[1 3 2 6 8 4]
```

NumPy ufunc :

- Universal functions.
- They work on **ndarray**.
- They are way faster than other functions



ufuncs also take additional arguments, like:

`where` boolean array or condition defining where the operations should take place.

`dtype` defining the return type of elements.

`out` output array where the return value should be copied.

-