# NumPy

*By Shreeyansh, Source: w3 Schools, NumPy Documentation*

---

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

---

## Why Numpy is faster than lists?

---

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Numpy is partially ritten in python but most of the parts that require fast computation are written in C and C++.

---

To use Numpy library, we need to import it first

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

In [2]:
```
print(np.__version__)
```

```
1.20.1
```

```
In [3]: my_list = [34,67,89,80,24,56]
```

To create an array we use the `np.array()` function.

```
In [4]: arr = np.array(my_list)                    #way 1 to create a numpy array - Pass a pr
```

```
In [5]: print(arr)
```

```
[34 67 89 80 24 56]
```

```
In [6]: type(arr)
```

```
Out[6]: numpy.ndarray
```

```
In [7]: arr1 = np.array([16,34,67,59,42])       #way 2 to create a numpy array - Manually en
        print(arr1)
```

```
[16 34 67 59 42]
```

```
In [8]: arr2 = np.array((34,78,44,89,36))       #way 3 to create a numpy array - Manually en
        print(arr2)
```

```
[34 78 44 89 36]
```

---

## 0. Basics

---

### `np.ones(size)`

Return arrays of Ones

```
In [9]: Q = np.ones((4,5))
        print(Q)
```

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

### `np.zeros(size)`

Return arrays of Zeros

```
In [10]: R = np.zeros((6,3))
         print(R)
```

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

## `np.empty(size)`

Return a new array of given shape and type, without initializing entries. `empty`, unlike `zeros`, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

In [11]:
```
S = np.empty((3, 4))
print(S)
```

```
[[1.01421523e-311 3.16202013e-322 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 8.37524822e+169 9.41531521e-067 1.04091413e-071]
 [2.22944852e+180 2.31786971e-052 3.11278513e+179 6.64087537e-066]]
```

## `np.eye(size)`

Return a 2-D array with ones on the diagonal and zeros elsewhere.

In [12]:
```
T = np.eye(4,5)
print(T)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]
```

## `np.identity(size)`

Return the identity array.

In [13]:
```
U = np.identity(5)
print(U)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

## `np.full(size, fill_value)`

Return a new array of given shape and type, filled with *fill_value*.

In [14]:
```
V = np.full((3,5),63.45)
print(V)
```

```
[[63.45 63.45 63.45 63.45 63.45]
 [63.45 63.45 63.45 63.45 63.45]
 [63.45 63.45 63.45 63.45 63.45]]
```

## Constants in NumPy

- `numpy.Inf` - IEEE 754 floating point representation of (positive) infinity.
- `numpy.Infinity` - IEEE 754 floating point representation of (positive) infinity.
- `numpy.NAN` - IEEE 754 floating point representation of Not a Number (NaN).NaN and NAN are equivalent definitions of nan. Please use nan instead of NAN.

- `numpy.NINF` - IEEE 754 floating point representation of negative infinity.
- `numpy.NZERO` - IEEE 754 floating point representation of negative zero.
- `numpy.PINF` - IEEE 754 floating point representation of (positive) infinity. Use inf because Inf, Infinity, PINF and infty are aliases for inf. For more details, see inf.
- `numpy.PZERO` - IEEE 754 floating point representation of positive zero.
- `numpy.e` - Euler's constant, base of natural logarithms, Napier's constant. e = 2.71828...
- `numpy.pi` - Constant pi = 3.1415926...

---

## 1. Dimensions in Arrays

---

- NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.
- NumPy arrays have an attribute `shape` that returns a tuple with each index having the number of corresponding elements.
- NumPy arrays have an attribute `reshape` that can change the shape of an array

#### Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.

We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

#### Unknown Dimension

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. *Note: We can not pass -1 to more than one dimension.*

### *Zero Dimensional Array*

In [15]:
```
Zero_Dim = np.array(42)
print(Zero_Dim)
```
```
42
```

In [16]:
```
print("Type of object",type(Zero_Dim))
print("#Dimensions = ",Zero_Dim.ndim)
print("Shape = ",Zero_Dim.shape)
```
```
Type of object <class 'numpy.ndarray'>
#Dimensions =  0
Shape =  ()
```

### *One Dimensional Array*

In [17]:
```
One_Dim = np.array([89,56,67,43,55,37])                                  #0 plane -
print(One_Dim)
```
```
[89 56 67 43 55 37]
```

In [18]:
```
print("Type of object",type(One_Dim))
print("#Dimensions = ",One_Dim.ndim)
print("Shape = ",One_Dim.shape)
```

```
Type of object <class 'numpy.ndarray'>
#Dimensions =  1
Shape =  (6,)
```

In [19]:
```
One_Dim.reshape(3,2)
```

Out[19]:
```
array([[89, 56],
       [67, 43],
       [55, 37]])
```

## *Two Dimensional Array*

In [20]:
```
Two_Dim = np.array([ [1,2,3],[4,5,6] ])                                          #1 pl
print(Two_Dim)
```

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

In [21]:
```
print("Type of object",type(Two_Dim))
print("#Dimensions = ",Two_Dim.ndim)
print("Shape = ",Two_Dim.shape)
```

```
Type of object <class 'numpy.ndarray'>
#Dimensions =  2
Shape =  (2, 3)
```

In [22]:
```
Two_Dim.reshape(6,1)
```

Out[22]:
```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

## *Three Dimensional Array*

In [23]:
```
Three_Dim = np.array([ [ [1,2,3],[4,5,6] ], [ [7,8,9],[10,11,12] ] ])      #2 planes
print(Three_Dim)                                                            #outer bra
```

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

 [[ 7  8  9]
  [10 11 12]]]
```

In [24]:
```
print("Type of object",type(Three_Dim))
print("#Dimensions = ",Three_Dim.ndim)
print("Shape = ",Three_Dim.shape)     # O/P -> (#planes, #rows, #columns)
```

```
Type of object <class 'numpy.ndarray'>
#Dimensions =  3
Shape =  (2, 2, 3)
```

## *Transposing an Array*

Reverse or permute the axes of an array; returns the modified array. For an array *a* with two axes, transpose(a) gives the matrix transpose.

In [25]:
```
O = np.ones((1,6))
print(O)
```

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

**Method 1 - Using `np.transpose(array)`**

In [26]:
```
np.transpose(O)
```

Out[26]:
```
array([[1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.]])
```

**Method 2 - Using `array.T`**

In [27]:
```
O.T
```

Out[27]:
```
array([[1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.]])
```

***NB***: Transposing a 1D array has no efect on its view. The variable O above is a matrix (notice the double square bracket). There are some ways to do it though,

- Pass the array **as a matrix** via using `np.matrix(1d array).T`

In [28]:
```
np.matrix([1,2,2,3,3,3,4,4,4,4,4]).T
```

Out[28]:
```
matrix([[1],
        [2],
        [2],
        [3],
        [3],
        [3],
        [4],
        [4],
        [4],
        [4],
        [4]])
```

- Pass the array as matrix in `np.transpose()` via using [array]

In [29]:
```
x = np.linspace(0,100,5)
np.transpose([x])
```

Out[29]:
```
array([[  0.],
       [ 25.],
       [ 50.],
       [ 75.],
       [100.]])
```

- Use `array.reshpe((-1,1))`

In [30]:
```
x.reshape((-1,1))
```

Out[30]:
```
array([[  0.],
       [ 25.],
       [ 50.],
       [ 75.],
       [100.]])
```

---

## 2. Flattening Array

---

Flattening array means converting a multidimensional array into a 1D array. We can use `reshape(-1)` to do this. There are a lot of functions for changing the shapes of arrays in numpy `flatten`, `ravel` and also for rearranging the elements `rot90`, `flip`, `fliplr`, `flipud` etc.

*To read the array in row/column major form there are certain orders to follow - '**C**', '**F**', '**A**', '**K**'*

'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran-style) order. 'A' means to flatten in column-major order if array is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten array in the order the elements occur in memory. The default is 'C'.

**Method 1 - Using `arr.reshape(-1)` : Elements read in row-major order**

In [31]:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])

arr_flattened = arr.reshape(-1)
print(arr_flattened)
```

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

**Method 2 - Using `np.ravel(array, order)`**

In [32]:
```
x = np.array([[1, 2, 3], [4, 5, 6],[7, 8, 9]])
print("Original Array: ")
print(x)

row_major = np.ravel(x, order = 'C')
print("Row Major Order: ",row_major)

column_major = np.ravel(x, order = 'F')
print("Column Major Order: ",column_major)
```

```
Original Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Row Major Order:  [1 2 3 4 5 6 7 8 9]
Column Major Order:  [1 4 7 2 5 8 3 6 9]
```

**Method 3 - Using `array.flatten(order)`**

In [33]:
```
flattened_array = x.flatten()    #default order is 'C'
print(flattened_array)
```

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

In [34]:
```
print(x.flatten('F'))
```

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

---

## 3. Array Indexing

---

Indexing is similar to that of C/C++, just, instead of writing each index in seperate brackets, use single bracket for all indices seperated by commas.

In [35]:
```
One_Dim[3]          #Element at 3rd index
```

Out[35]:
```
43
```

In [36]:
```
Two_Dim[1,2]       #Element at 2nd Row (1), 3rd column(2) (indexing of each dimensio
```

Out[36]:
```
6
```

In [37]:
```
Three_Dim[0,1,2]   #Element at 1st plane (0), 2nd Row (1), 3rd Column (2)
```

Out[37]:
```
6
```

***Negative Indexing is used to access an array from the end***

In [38]:
```
Two_Dim[1,-1]     #Last element at 2nd row
```

Out[38]:
```
6
```

---

## 4. Array Slicing

---

Slicing in python means taking elements from one given index to another given index.We pass slice instead of index like this: `[start:end].` We can also define the step, like this: `[start:end:step].`

- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension
- If we don't pass step its considered 1

***The result includes the start index, but excludes the end index.***

In [39]:
```
print(One_Dim[1:4])   #1st Index to 3rd Index
```

```
[56 67 43]
```

In [40]:
```
print(Two_Dim[1, 1:3]) #2nd Row --> elements 1st and 2nd
```
```
[5 6]
```

In [41]:
```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print("Original:")
print(arr)
print("\r")
print("Sliced: ",arr[0:2, 2])   #From both rows (0:2), return 2nd index
```
```
Original:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]

Sliced:  [3 8]
```

In [42]:
```
arr2 = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr2[0:2, 1:4])  #From both rows, return 1st to 3rd indices
```
```
[[2 3 4]
 [7 8 9]]
```

---

## 5. Datatypes in NumPy

---

NumPy has some extra data types, and refer to data types with one character, like `i` for integers, `u` for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- `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 )

The NumPy array object has a property called `dtype` that returns the data type of the array:

In [43]:
```
print(Three_Dim.dtype)
```
```
int32
```

We use the `array()` function to create arrays, this function can take an optional argument: `dtype` that allows us to define the expected data type of the array elements:

In [44]:
```
arr = np.array([1, 2, 3, 4], dtype='S')
```

```
print(arr)
print(arr.dtype)
```

```
[b'1' b'2' b'3' b'4']
|S1
```

For `i`, `u`, `f`, `S` and `U` we can define size as well.

In [45]:

```
arr = np.array([1, 2, 3, 4], dtype='i4')  #Create an array with data type 4 byte int

print(arr)
print(arr.dtype)
```

```
[1 2 3 4]
int32
```

### *Converting datatype on existing array*

The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.

The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like `f` for float, `i` for integer etc. or you can use the data type directly like float for float and int for integer.

In [46]:

```
arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

---

## 6. Copy vs View

---

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

- The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy. The copy SHOULD NOT be affected by the changes made to the original array.

- The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view. The view SHOULD be affected by the changes made to the original array.

In [47]:

```
alias_copy = One_Dim.copy()
print(alias_copy)
```

```
[89 56 67 43 55 37]
```

In [48]:
```
print("Original's Address: ",id(One_Dim))
print("Copy's Address: ",id(alias_copy))
```

```
Original's Address:  2052809110992
Copy's Address:  2052809279504
```

In [49]:
```
alias_view = One_Dim.view()
print(alias_view)
```

```
[89 56 67 43 55 37]
```

In [50]:
```
print("Original's Address: ",id(One_Dim))
print("View's Address: ",id(alias_view))
```

```
Original's Address:  2052809110992
View's Address:  2052809281232
```

In [51]:
```
alias_view[1] = 23
print(alias_view)
print(One_Dim)   #Original Object Changed if view changed
```

```
[89 23 67 43 55 37]
[89 23 67 43 55 37]
```

In [52]:
```
alias_copy[2] = 23
print(alias_copy)
print(One_Dim)     #Original Object Unchanged if copy changed
```

```
[89 56 23 43 55 37]
[89 23 67 43 55 37]
```

### *Check if Array Owns Its's Data*

As mentioned above, copies owns the data, and views does not own the data, but how can we check this? Every NumPy array has the attribute `base` that returns `None` if the array owns the data. Otherwise, the base attribute refers to the original object.

In [53]:
```
arr = np.array([1, 2, 3, 4, 5])

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

```
None
[1 2 3 4 5]
```

In [54]:
```
x.reshape(5,1).base
```

Out[54]:
```
array([1, 2, 3, 4, 5])
```

---

## 7. Iterating over arrays

---

The function `nditer()` is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration.

- **Iterating on Each Scalar Element** - In basic 'for' loops, iterating through each scalar of an array we need to use *n* 'for' loops for *n*-dimensional array which can be difficult to write for arrays with very high dimensionality.

In [55]:
```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

for x in np.nditer(arr):
  print(x)
```

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

- **Iterating Array With Different Data Types** - We can use `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'].`

In [56]:
```
arr = np.array([1, 2, 3])

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

```
b'1'
b'2'
b'3'
```

- **Iterating With Different Step Size** - We can use filtering and followed by iteration.

In [57]:
```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for x in np.nditer(arr[:, ::2]):     #Iterate with step of 2
  print(x)
```

```
1
3
5
7
```

- **Enumerated Iteration Using `ndenumerate()`** - Sometimes we require corresponding index of the element while iterating, the `ndenumerate()` method can be used for those usecases.

In [58]:
```
arr = np.array([1, 2, 3])

for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

```
(0,) 1
(1,) 2
(2,) 3
```

## 8. Joining Arrays

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes. We pass a sequence of arrays that we want to join to the `concatenate()` function, along with the axis. If axis is not explicitly passed, it is taken as 0.

`axis = 0` - row wise

`axis = 1` - column wise

In [59]:

```
from numpy import random
arr1 = random.randint(100, size=(3,5))
arr2 = random.randint(100, size=(3,5))

arr3 = np.concatenate((arr1,arr2))
print(arr1)
print("\r")
print(arr2)
print("\r")
print("Joined: \r")
print(arr3)
```

```
[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]]

[[82 82 19 70 78]
 [59 33 70 38 82]
 [85 46 12 60 60]]

Joined:
[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]
 [82 82 19 70 78]
 [59 33 70 38 82]
 [85 46 12 60 60]]
```

- **Stack Functions**

Stacking is same as concatenation, the only difference is that stacking is done along a new axis. We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking. We pass a sequence of arrays that we want to join to the `stack()` method along with the axis. If axis is not explicitly passed it is taken as 0.

In [60]:

```
arr3 = np.stack((arr1,arr2), axis = 0)
print(arr3)
```

```
[[[69 80 89 96 42]
  [88 56 70 93 53]
  [36 24 17 64 63]]

 [[82 82 19 70 78]
 [59 33 70 38 82]
 [85 46 12 60 60]]]
```

In [61]:
```
arr3 = np.stack((arr1,arr2), axis = 1)
print(arr3)
```

```
[[[69 80 89 96 42]
  [82 82 19 70 78]]

 [[88 56 70 93 53]
  [59 33 70 38 82]]

 [[36 24 17 64 63]
  [85 46 12 60 60]]]
```

- **Stacking Along Rows** -

NumPy provides a helper function: `hstack()` to stack along rows.

In [62]:
```
arr3 = np.hstack((arr1,arr2))
print(arr3)


# -----     -----     ----- -----
# |   |  +  |   |  =  |   | |   |
# |   |     |   |     |   | |   |
# -----     -----     ----- -----
```

```
[[69 80 89 96 42 82 82 19 70 78]
 [88 56 70 93 53 59 33 70 38 82]
 [36 24 17 64 63 85 46 12 60 60]]
```

- **Stacking Along Columns** -

NumPy provides a helper function: `vstack()` to stack along columns.

In [63]:
```
arr4 = np.vstack((arr1,arr2))
print(arr4)

# -----     -----     -----
# |   |  +  |   |  =  |   |
# |   |     |   |     |   |
# -----     -----     -----
#                     -----
#                     |   |
#                     |   |
#                     -----
```

```
[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]
 [82 82 19 70 78]
 [59 33 70 38 82]
 [85 46 12 60 60]]
```

- **Stacking Along Depth/Height**

NumPy provides a helper function: `dstack()` to stack along height, which is the same as depth.

In [64]:

```
arr5 = np.dstack((arr1,arr2))
print(arr1)
print("\r")
print(arr2)
print("\r\r")
print(arr5)
```

```
[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]]

[[82 82 19 70 78]
 [59 33 70 38 82]
 [85 46 12 60 60]]

[[[69 82]
  [80 82]
  [89 19]
  [96 70]
  [42 78]]

 [[88 59]
  [56 33]
  [70 70]
  [93 38]
  [53 82]]

 [[36 85]
  [24 46]
  [17 12]
  [64 60]
  [63 60]]]
```

---

## 9. Splitting Array

---

Splitting is reverse operation of Joining. Joining merges multiple arrays into one and Splitting breaks one array into multiple. We use

`array_split(array, #parts, axis)` for splitting arrays, we pass it the array we want to split and the number of splits. `axis` is optional argument whose value by default is 0.

In [65]:
```
x = np.array_split(arr1, 3)
print(arr1)
print("\r\r")
print(x)
```

```
[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]]

[array([[69, 80, 89, 96, 42]]), array([[88, 56, 70, 93, 53]]), array([[36, 24, 17, 6
4, 63]])]
```

**If the array has less elements than required, it will adjust from the end accordingly.**

In [66]:
```
x = np.array_split(arr1, 4)
print(arr1)
print("\r\r")
print(x)
```

```
[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]]

[array([[69, 80, 89, 96, 42]]), array([[88, 56, 70, 93, 53]]), array([[36, 24, 17, 6
4, 63]]), array([], shape=(0, 5), dtype=int32)]
```

In [67]:
```
print(x[0])
print(x[1])
print(x[2])
print(x[3])
```

```
[[69 80 89 96 42]]
[[88 56 70 93 53]]
[[36 24 17 64 63]]
[]
```

*N.B.:We also have the method `split()` available but it will not adjust the elements when elements are less in source array for splitting like in example above, `array_split()` worked properly but `split()` would fail.*

In [68]:
```
print("Before Split:\r")
print(arr4)                                     #6 rows
y = np.array_split(arr4,2)                      #split in 2 parts therefore 3 - 3 pair
print("After Split:\r")
print(y)
print("y[0] =\r\r")
print(y[0])
print("y[1] = \r\r")
print(y[1])
```

```
Before Split:
[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]
 [82 82 19 70 78]
 [59 33 70 38 82]
 [85 46 12 60 60]]
After Split:
[array([[69, 80, 89, 96, 42],
       [88, 56, 70, 93, 53],
       [36, 24, 17, 64, 63]]), array([[82, 82, 19, 70, 78],
       [59, 33, 70, 38, 82],
       [85, 46, 12, 60, 60]])]

[[69 80 89 96 42]
 [88 56 70 93 53]
 [36 24 17 64 63]]

[[82 82 19 70 78]
 [59 33 70 38 82]
 [85 46 12 60 60]]
```

**In addition, you can specify which axis you want to do the split around.**

Split along rows, `axis = 0` Split along columns, `axis = 1`

In [69]:
```
Q = random.randint(100, size = (4,4))
print(Q)
print("Row Split\r\r")
L = np.array_split(Q, 2, axis = 0)
print(L)
print("\r\r")
print(L[0])
print(L[1])
```

```
[[50 65 84 37]
 [10 88 58 79]
 [51 83 21 24]
 [78 60 60 60]]
Row Split
[array([[50, 65, 84, 37],
       [10, 88, 58, 79]]), array([[51, 83, 21, 24],
       [78, 60, 60, 60]])]

[[50 65 84 37]
 [10 88 58 79]]
[[51 83 21 24]
 [78 60 60 60]]
```

In [70]:

```
print(Q)
print("Column Split\r\r")
M = np.array_split(Q, 2, axis = 1)
print(M)
print("\r\r")
print(M[0])
print(M[1])
```

```
[[50 65 84 37]
 [10 88 58 79]
 [51 83 21 24]
 [78 60 60 60]]

[array([[50, 65],
       [10, 88],
       [51, 83],
       [78, 60]]), array([[84, 37],
       [58, 79],
       [21, 24],
       [60, 60]])]

[[50 65]
 [10 88]
 [51 83]
 [78 60]]
[[84 37]
 [58 79]
 [21 24]
 [60 60]]
```

### `hsplit()` and `vsplit()`

- `hsplit()` - Split an array into multiple sub-arrays horizontally (column-wise).
- `vsplit()` - Split an array into multiple sub-arrays vertically (row-wise).

**Try Not to use this one, use `np.split(array, #parts, axis)` instead**

In [71]:

```
print(Q)
A = np.hsplit(Q,2)
print(A)
print(A[0])
print(A[1])
```

```
[[50 65 84 37]
 [10 88 58 79]
 [51 83 21 24]
 [78 60 60 60]]
[array([[50, 65],
       [10, 88],
       [51, 83],
       [78, 60]]), array([[84, 37],
       [58, 79],
       [21, 24],
       [60, 60]])]
[[50 65]
 [10 88]
 [51 83]
 [78 60]]
[[84 37]
 [58 79]
 [21 24]
 [60 60]]
```

In [72]:

```
print(Q)
A = np.vsplit(Q,2)
print(A)
print(A[0])
print(A[1])
```

```
[[50 65 84 37]
 [10 88 58 79]
 [51 83 21 24]
 [78 60 60 60]]
[array([[50, 65, 84, 37],
       [10, 88, 58, 79]]), array([[51, 83, 21, 24],
       [78, 60, 60, 60]])]
[[50 65 84 37]
 [10 88 58 79]]
[[51 83 21 24]
 [78 60 60 60]]
```

---

## 10. Searching

---

You can search an array for a certain value, and return the indexes that get a match. To search an array, use the `where()` method.

In [73]:

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

x = np.where(arr == 4)    #returns a tuple of indices indicating where the search el

print(x)
```

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

### `searchsorted()`

There is a method called `searchsorted()` which performs a binary search in the array, and returns the index **where the specified value would be inserted to maintain the search order.**

***The `searchsorted()` method is assumed to be used on sorted arrays.***

In [74]:

```
arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)
print(x)
```

```
1
```

### Searching for Multiple Values

To search for more than one value, use an array with the specified values.

In [75]:
```
arr = np.array([1, 3, 5, 7])

x = np.searchsorted(arr, [2, 4, 6])

print(x)
```

```
[1 2 3]
```

The return value is an array: [1 2 3] containing the three indexes where 2, 4, 6 would be inserted in the original array to maintain the order.

---

# 11. Sorting an array

---

Sorting means putting elements in an ordered sequence.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called `sort()`, that will sort a specified array. ***Note: This method returns a copy of the array, leaving the original array unchanged.***

In [76]:
```
arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

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

In [77]:
```
arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

In [78]:
```
arr = np.array([[3, 2, 4], [5, 0, 1]])

print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

---

# 12. NumPy Filter Array

---

Getting some elements out of 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 which 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.

In [79]:

```
arr = np.array([41, 42, 43, 44])

x = [True, False, True, False]

newarr = arr[x]

print(newarr)
```

```
[41 43]
```

## Creating a Filter array

In the example above we hard-coded the `True` and `False` values, but the common use is to create a filter array based on conditions.

In [80]:

```
arr = np.array([41, 42, 43, 44])

# Create an empty list
filter_arr = []

# go through each element in arr
for element in arr:
  # if the element is higher than 42, set the value to True, otherwise False:
  if element > 42:
    filter_arr.append(True)
  else:
    filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

```
[False, False, True, True]
[43 44]
```

## Creating Filter array directly from array

The above example is quite a common task in NumPy and NumPy provides a nice way to tackle it.

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

In [81]:

```
#Creating a filter array that will return true iff element>42
arr = np.array([41, 42, 43, 44])

filter_arr = arr > 42

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

```
[False False  True  True]
[43 44]
```

---

# 13. Miscellaneous Functions

---

## `np.arange()` - Return evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list.

Syntax: `np.arange(start,stop,step)`

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `numpy.linspace` for these cases.

In [82]:

```python
import numpy as np
x = np.arange(10) #with only 1 argument, its assumed as end value with start from 0
y = np.arange(0,10) #with only 2 arguments, these are assumed as start and end value
z = np.arange(0,10,2)
print(x,y,z)
```

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

## `np.linspace()` - Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the interval `[start, stop]`. The endpoint of the interval can optionally be excluded. ***NOTE THAT ITS `LIN` NOT `LINE`***

In [83]:

```python
np.linspace(2,3, num = 5)
```

Out[83]:

```
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

## `np.broadcast()` - Produce an object that mimics broadcasting

In [84]:

```python
x = np.array([[1], [2], [3]])
y = np.array([4, 5, 6])
b = np.broadcast(x, y)
print(b)
```

```
<numpy.broadcast object at 0x000001DDF42D2CE0>
```

In [85]:

```python
out = np.empty(b.shape)
out.flat = [u+v for (u,v) in b]
out
```

Out[85]:

```
array([[5., 6., 7.],
       [6., 7., 8.],
       [7., 8., 9.]])
```

*Compare against built-in broadcasting*

In [86]:
```
x+y
```

Out[86]:
```
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

### `np.squeeze()` - Remove axes of length one from array.

In [87]:
```
x = np.array([[[0], [1], [2]]])
x.shape
```

Out[87]:
```
(1, 3, 1)
```

In [88]:
```
np.squeeze(x)
```

Out[88]:
```
array([0, 1, 2])
```

In [89]:
```
np.squeeze(x).shape
```

Out[89]:
```
(3,)
```

In [90]:
```
print(Q)
```

```
[[50 65 84 37]
 [10 88 58 79]
 [51 83 21 24]
 [78 60 60 60]]
```

In [91]:
```
np.squeeze(Q)
```

Out[91]:
```
array([[50, 65, 84, 37],
       [10, 88, 58, 79],
       [51, 83, 21, 24],
       [78, 60, 60, 60]])
```

In [92]:
```
np.squeeze(Q).shape
```

Out[92]:
```
(4, 4)
```

In [93]:
```
Q.shape
```

Out[93]:
```
(4, 4)
```

### `np.delete()` - Return a new array with sub-arrays along an axis deleted.

For a one dimensional array, this returns those entries not returned by arr[obj].

`numpy.delete(arr, idx, axis=None)`

Parameters

1. **arr: array_like** - Input array.

2. **obj/idx: slice, int or array of ints** - Indicate indices of sub-arrays to remove along the specified axis.

1. **axis: int, optional** - The axis along which to delete the subarray defined by obj. If axis is None, obj is applied to the flattened array.

```
In [94]:
arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
arr
```

```
Out[94]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])
```

```
In [95]:
np.delete(arr, 1, axis = 0)  # 1: remove arr[1], i.e, 2nd row
```

```
Out[95]: array([[ 1,  2,  3,  4],
                [ 9, 10, 11, 12]])
```

```
In [96]:
np.delete(arr, np.s_[::2], 1)
```

```
Out[96]: array([[ 2,  4],
                [ 6,  8],
                [10, 12]])
```

```
In [97]:
np.delete(arr, [1,3,5], axis = None)
```

```
Out[97]: array([ 1,  3,  5,  7,  8,  9, 10, 11, 12])
```

## `np.insert()` - Insert values along the given axis before the given indices.

`numpy.insert(arr, idx, values, axis=None)`

Parameters

1. **arr: array_like** - Input array.

2. **obj/idx: int**, slice or sequence of ints - Object that defines the index or indices before which values is inserted. Support for multiple insertions when obj is a single scalar or a sequence with one element (similar to calling insert multiple times).

3. **values: *array_like*** - Values to insert into arr. If the type of values is different from that of arr, values is converted to the type of arr. values should be shaped so that arr[...,obj,...] = values is legal.

4. **axis: int**, optional - Axis along which to insert values. If axis is None then arr is flattened first.

***Original array remains unchanged. To keep the changes, assign the modified array to a new object.***

In [98]:
```
trance = np.array([ [1,2,3],[4,5,6],[7,8,9] ])
trance
```

Out[98]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [99]:
```
y = np.insert(trance, 3, [10,11,12], axis = 0 )
y
```

Out[99]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [100…
```
np.insert(y, 3, [11,22,33,44], axis = 1)
```

Out[100…
```
array([[ 1,  2,  3, 11],
       [ 4,  5,  6, 22],
       [ 7,  8,  9, 33],
       [10, 11, 12, 44]])
```

In [101…
```
x = np.arange(8).reshape(2, 4)
idx = (1, 3)
np.insert(x, idx, 999, axis=1)   # 999 broadcasted to all rows
```

Out[101…
```
array([[  0, 999,   1,   2, 999,   3],
       [  4, 999,   5,   6, 999,   7]])
```

## `np.append()` - Append values to the end of an array

`numpy.append(arr, values, axis=None)`

Parameters

1. **arr: array_like** - Input array.
2. **values: *array_like*** - Values to insert into arr. If the type of values is different from that of arr, values is converted to the type of arr. values should be shaped so that arr[...,obj,...] = values is legal.
3. **axis: int**, optional - Axis along which to insert values. Note that append does not occur in-place: a new array is allocated and filled. If *axis* is None then arr is flattened first.

***Original array remains unchanged. To keep the changes, assign the modified array to a new object.***

In [102…
```
y
```

Out[102…
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

When `axis` is specified, values must have the correct shape. Keep track of brackets

```
In [103…  np.append(y, [13,14,15], axis = 0) #valueError
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-103-b0f1cf7dc811> in <module>
----> 1 np.append(y, [13,14,15], axis = 0) #valueError

<__array_function__ internals> in append(*args, **kwargs)

C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\function_base.py in append(arr,
values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746
   4747

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input arrays must have same number of dimensions, but the array
 at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)
```

```
In [104…  k = np.append(y,[ [13,14,15] ], axis = 0)
          k
```

```
Out[104…  array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [10, 11, 12],
                 [13, 14, 15]])
```

```
In [105…  np.append(k ,[20,21,22,23,24], axis = 1) #valueError
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-105-de376e72d733> in <module>
----> 1 np.append(k ,[20,21,22,23,24], axis = 1) #valueError

<__array_function__ internals> in append(*args, **kwargs)

C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\function_base.py in append(arr,
values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746
   4747

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input arrays must have same number of dimensions, but the array
 at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)
```

```
In [106…  np.append(k ,[[20,21,22,23,24]], axis = 1) #valueError
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-106-e107e926d313> in <module>
----> 1 np.append(k ,[[20,21,22,23,24]], axis = 1) #valueError

<__array_function__ internals> in append(*args, **kwargs)

C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\function_base.py in append(arr,
values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746 
   4747 

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 0, the array at index 0 has size 5 and the array at index 1 has size 1
```

In [107…

```
np.append(k ,[[20],[21],[22],[23],[24]], axis = 1)
```

Out[107…

```
array([[ 1,  2,  3, 20],
       [ 4,  5,  6, 21],
       [ 7,  8,  9, 22],
       [10, 11, 12, 23],
       [13, 14, 15, 24]])
```

## `np.flip()` - Reverse the order of elements in an array along the given axis.

The shape of the array is preserved, but the elements are reordered.

In [108…

```
Q
```

Out[108…

```
array([[50, 65, 84, 37],
       [10, 88, 58, 79],
       [51, 83, 21, 24],
       [78, 60, 60, 60]])
```

In [109…

```
np.flip(Q, axis=0)
```

Out[109…

```
array([[78, 60, 60, 60],
       [51, 83, 21, 24],
       [10, 88, 58, 79],
       [50, 65, 84, 37]])
```

In [110…

```
np.flip(Q, axis=1)
```

Out[110…

```
array([[37, 84, 65, 50],
       [79, 58, 88, 10],
       [24, 21, 83, 51],
       [60, 60, 60, 78]])
```

- `flip(m, 0)` is equivalent to `flipud(m)`.
- `flip(m, 1)` is equivalent to `fliplr(m)`.
- `flip(m, n)` corresponds to m[...,::-1,...] with ::-1 at position n.
- `flip(m)` corresponds to m[::-1,::-1,...,::-1] with ::-1 at all positions.
- `flip(m, (0, 1))` corresponds to m[::-1,::-1,...] with ::-1 at position 0 and position 1.