WHY USE NUMPY?

l1 = [1, 2, 3] ← C

- NumPy provides efficient storage
- It also provides better ways of handling data for processing
- It is fast
- It is easy to learn
- NumPy uses relatively less memory to store data

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

In [2]:
```
arr = np.array([1,2,3,4,5], np.int32) #Can do np.int8/16/32/62
                                      #Or np.float32/64
```

In [3]:
```
type(arr)
```

Out[3]:
```
numpy.ndarray
```

In [4]:
```
myarr = np.array([[1,3,2,6,7,4]]) #2D array
```

In [5]:
```
myarr[0,2]
```

Out[5]:
```
2
```

In [6]:
```
myarr.dtype #Default dtype is int64
```

Out[6]:
```
dtype('int64')
```

In [7]:
```
myarr.size
```

Out[7]:
```
6
```

In [8]:
```
myarr.shape
```

Out[8]:
```
(1, 6)
```

In [9]:
```
myarr[0,1] = 10 #Changing value of element (0,1)
```

In [10]:
```
myarr
```

Out[10]:
```
array([[ 1, 10,  2,  6,  7,  4]])
```

Making numpy array from list/tuple

In [11]:
```
list1 = [1,22,3,4,59,6,10]
```

In [12]:
```
arr1 = np.array(list1) #1D Array
arr1
```

Out[12]:
```
array([ 1, 22,  3,  4, 59,  6, 10])
```

In [13]:
```
arr1.dtype #As my system is 64bit so default dtype is 64 bit
```

Out[13]:
```
dtype('int64')
```

In [14]:
```
list2 = [[1,2,3], [2,3,11], [34,5,21]]
```

In [15]:
```
arr2 = np.array(list2, np.int16) #2D Array
arr2
```

Out[15]:
```
array([[ 1,  2,  3],
       [ 2,  3, 11],
       [34,  5, 21]], dtype=int16)
```

In [16]:
```
arr2.dtype #Here i specified the dtype to be int16
```

Out[16]:
```
dtype('int16')
```

In [ ]:
```

```

In [17]:
```
tuple1 = (1,32,45,23,46)
```

In [18]:
```
arr3 = np.array(tuple1, np.float32)
```

In [19]:
```
arr3.dtype
```

Out[19]:
```
dtype('float32')
```

In [ ]:
```

```

In [20]:
```
dict1 = {1,2,45}
```

In [21]:
```
np.array(dict1) #dtype of the array created using dict is object, which is
```

Out[21]:
```
array({1, 2, 45}, dtype=object)
```

In [ ]:

Intrinsic numpy array creation objects

In [22]:
```
zeros = np.zeros((2,2))
zeros
```

Out[22]:
```
array([[0., 0.],
       [0., 0.]])
```

In [23]:
```
zeros.dtype
```

Out[23]:
```
dtype('float64')
```

In [24]:
```
zeros = np.zeros((2,2), np.int32)
zeros
```

Out[24]:
```
array([[0, 0],
       [0, 0]], dtype=int32)
```

In [ ]:

In [25]:
```
rng = np.arange(11) #creates numpy array from 0 to n-1
rng
```

Out[25]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [ ]:

In [26]:
```
lspace = np.linspace(1, 50, 10) # from 1 to 50 divided in to give equal sp
lspace
```

Out[26]:
```
array([ 1.        ,  6.44444444, 11.88888889, 17.33333333, 22.77777778,
       28.22222222, 33.66666667, 39.11111111, 44.55555556, 50.        ])
```

In [27]:
```
lspace.size
```

Out[27]:
```
10
```

In [ ]:

In [28]:
```
emp = np.empty((3,3)) #Gives a numpy array of 3*3 size with all random val
emp
```

Out[28]:
```
array([[ 2.60605835e-31, -5.21211670e-31,  1.30302917e-31],
       [-5.21211670e-31,  1.13363538e-30, -3.51817877e-31],
       [ 1.30302917e-31, -3.51817877e-31,  2.01969522e-31]])
```

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

Out[29]:
```
array([ 1.        ,  6.44444444, 11.88888889, 17.33333333, 22.77777778,
       28.22222222, 33.66666667, 39.11111111, 44.55555556, 50.        ])
```

In [ ]:

Identity Matrix

In [30]:
```
ide = np.identity(5) # Gives identity matrix of 5*5
ide
```

Out[30]:
```
array([[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.]])
```

In [31]:
```
ide.shape
```

Out[31]:
```
(5, 5)
```

In [ ]:

In [32]:
```
arr = np.arange(99)
arr
```

Out[32]:
```
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 [33]:
```
arr.size
```

Out[33]:
```
99
```

In [34]:
```
arr.reshape(3, 33) # As arr is of 99 size, so if we reshape it then n*m = 
                   # eg: 3*33 = 99, 9*11 = 99 etc..
```

Out[34]:
```
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 [35]:
```
arr = arr.reshape(9, 11)
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]])
```

In [36]:
```
arr.shape
```

Out[36]:
```
(9, 11)
```

In [ ]:
```

```

To flatten

In [37]:
```
arr = arr.flatten() # Can also use arr = arr.flatten()
arr
```

Out[37]:
```
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 [38]:
```
arr.shape
```

Out[38]:
```
(99,)
```

In [ ]:
```

```

In [ ]:
```
```

In [39]:
```
ar = np.array([[1,2,3],
               [4,5,6],
               [7,8,9]])
ar
```

Out[39]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [40]:
```
ar.sum(axis = 0) #Sum in vertical direction
                 #Axis = 0 is for rows
```

Out[40]:
```
array([12, 15, 18])
```

In [41]:
```
ar.sum(axis = 1) #Sum in horizontal direction
                 #Axis = 1 is for columns
```

Out[41]:
```
array([ 6, 15, 24])
```

In [ ]:
```
```

Transpose

In [42]:
```
ar.T # To change the original array ar = ar.T
```

Out[42]:
```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

In [43]:
```
ar.flat #Returns as an iterator
```

Out[43]:
```
<numpy.flatiter at 0x7fed140e2800>
```

In [44]:
```
for item in ar.flat:
    print(item)
```

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

In [ ]:
```
```

In [45]:
```
ar
```

Out[45]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [46]:
```
ar.ndim #As ar is an 2D array
```

Out[46]:
```
2
```

In [47]:
```
priya = np.array([[[[1,2,3,4]]]])
```

In [48]:
```
priya.ndim #As priya is a 4D array
```

Out[48]:
```
4
```

In [ ]:
```
```

In [49]:
```
ar.nbytes #Tells how much bytes is comsumed by array
```

Out[49]:
```
72
```

In [50]:
```
one = np.array([21,34,22,100,34,12,3,4,5])
```

In [51]:
```
max_index = one.argmax() #returns the index of biggest element
```

In [52]:
```
one[max_index]
```

Out[52]:
```
100
```

In [ ]:
```
```

In [53]:
```
one.argsort() #Return the index order needed to sort the array
```

Out[53]:
```
array([6, 7, 8, 5, 0, 2, 1, 4, 3])
```

In [ ]:
```
```

In [ ]:
```
```

In [54]:
```
priya = np.array([[1,56,32],
                  [99,2,7],
                  [3,50,77]])
priya
```

Out[54]:
```
array([[ 1, 56, 32],
       [99,  2,  7],
       [ 3, 50, 77]])
```

In [55]:
```
priya.argmin() #It first flattens the array and then tells the index of sm
```

Out[55]:
```
0
```

In [56]:
```
priya.argmax() #It first flattens the array and then tells the index of la
```

Out[56]:
```
3
```

In [57]:
```
priya.argmin(axis = 0) #axis = 0 for rows
                       #Vertical
```

Out[57]:
```
array([0, 1, 1])
```

In [58]:
```
priya.argmin(axis = 1)
```

Out[58]:
```
array([0, 1, 0])
```

In [59]:
```
priya.argmax(axis = 0)
```

Out[59]:
```
array([1, 0, 2])
```

In [60]:
```
priya.argmax(axis = 1)
```

Out[60]:
```
array([1, 0, 2])
```

In [ ]:
```
```

In [61]:
```
priya
```

Out[61]:
```
array([[ 1, 56, 32],
       [99,  2,  7],
       [ 3, 50, 77]])
```

In [62]:
```
priya.argsort(axis = 0)
```

Out[62]:
```
array([[0, 1, 1],
       [2, 2, 0],
       [1, 0, 2]])
```

In [63]:
```
priya.argsort(axis = 1)
```

Out[63]:
```
array([[0, 2, 1],
       [1, 2, 0],
       [0, 1, 2]])
```

In [ ]:
```

```

In [64]:
```
ar1 = np.array([[1,1,1],
                [2,2,2],
                [3,3,3]])
```

In [65]:
```
ar2 = np.array([[10,10,10],
                [20,20,20],
                [30,30,30]])
```

In [66]:
```
ar1 + ar2 #If we did this in list then it would have concatenated the list
```

Out[66]:
```
array([[11, 11, 11],
       [22, 22, 22],
       [33, 33, 33]])
```

In [67]:
```
ar1 * ar2 #This is not matrix multiplcation, it is corresponding element m
```

Out[67]:
```
array([[10, 10, 10],
       [40, 40, 40],
       [90, 90, 90]])
```

In [ ]:
```

```

In [68]:
```
ar.max()
```

Out[68]:
```
9
```

In [69]:
```
ar.min()
```

Out[69]:
```
1
```

In [70]:
```
np.sqrt(ar1)
```

Out[70]:
```
array([[1.        , 1.        , 1.        ],
       [1.41421356, 1.41421356, 1.41421356],
       [1.73205081, 1.73205081, 1.73205081]])
```

In [71]:
```
np.power(ar1, 2)
```

Out[71]:
```
array([[1, 1, 1],
       [4, 4, 4],
       [9, 9, 9]])
```

In [ ]:
```

```

In [72]:
```
np.where(ar1>5)
```

Out[72]:
```
(array([], dtype=int64), array([], dtype=int64))
```

In [ ]:
```

```

Numpy array and python array size comparison

In [73]:
```
import sys
```

In [74]:
```
py_arr = [0,4,55,2]
```

In [75]:
```
sys.getsizeof(1) * len(py_arr)
```

Out[75]:
```
112
```

In [ ]:
```

```

In [76]:
```
np_arr = np.array([0,4,55,2])
```

In [77]:
```
np_arr.itemsize * np_arr.size
```

Out[77]:
```
32
```

In [ ]:

Convert Numpy array to list

In [78]:
```
np_arr.tolist()
```

Out[78]:
```
[0, 4, 55, 2]
```

In [ ]:

Creating a NumPy Array

Basic ndarray

NumPy arrays are very easy to create given the complex problems they solve. To create a very basic ndarray, you use the np.array() method. All you have to pass are the values of the array as a list:

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

Output:

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

This array contains integer values. You can specify the type of data in the **dtype** argument:

```
np.array([1,2,3,4],dtype=np.float32)
```

Output:

```
array([1., 2., 3., 4.], dtype=float32)
```

Since NumPy arrays can contain only homogeneous datatypes, values will be upcast if the types do not match:

```
np.array([1,2.0,3,4])
```

Output:

```
array([1., 2., 3., 4.])
```

Here, NumPy has upcast integer values to float values.

NumPy arrays can be multi-dimensional too.

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

array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Here, we created a 2-dimensional array of values.

Note: A matrix is just a rectangular array of numbers with shape N x M where N is the number of rows and M is the number of columns in the matrix. The one you just saw above is a 2 x 4 matrix.

Array of zeros

NumPy lets you create an array of all zeros using the **np.zeros()** method. All you have to do is pass the shape of the desired array:

```
np.zeros(5)

array([0., 0., 0., 0., 0.])
```

The one above is a 1-D array while the one below is a 2-D array:

```
np.zeros((2,3))

array([[0., 0., 0.],
       [0., 0., 0.]])
```

Array of ones

You could also create an array of all **1s** using the **np.ones()** method:

```
np.ones(5,dtype=np.int32)

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

Random numbers in ndarrays

Another very commonly used method to create ndarrays is np.random.rand() method. It creates an array of a given shape with random values from [0,1):

```
# random
np.random.rand(2,3)

array([[0.95580785, 0.98378873, 0.65133872],
       [0.38330437, 0.16033608, 0.13826526]])
```

An array of your choice

Or, in fact, you can create an array filled with any given value using the **np.full()** method. Just pass in the shape of the desired array and the value you want:

```
np.full((2,2),7)

array([[7, 7],
```

```
        [7, 7]])
```

Imatrix in NumPy

Another great method is **np.eye()** that returns an array with **1s** along its diagonal and **0s** everywhere else.

*An **Identity matrix** is a square matrix that has **1s along its main diagonal and 0s everywhere else**. Below is an Identity matrix of shape 3 x 3.*

Note: A square matrix has an N x N shape. This means it has the same number of rows and columns.

```
# identity matrix
np.eye(3)

array([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

However, NumPy gives you the flexibility to change the diagonal along which the values have to be **1s**. You can either move it above the main diagonal:

```
# not an identity matrix

np.eye(3,k=1)

array([[0., 1., 0.],
        [0., 0., 1.],
        [0., 0., 0.]])
```

Or move it below the main diagonal:

```
np.eye(3,k=-2)

array([[0., 0., 0.],
        [0., 0., 0.],
        [1., 0., 0.]])
```

Note: A matrix is called the Identity matrix only when the 1s are along the main diagonal and not any other diagonal!

Evenly spaced ndarray

You can quickly get an evenly spaced array of numbers using the **np.arange()** method:

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

The start, end and step size of the interval of values can be explicitly defined by passing in three numbers as arguments for these values respectively. A point to be noted here is that the interval is defined as [start,end) where the last number will not be included in the array:

```
np.arange(2,10,2)

array([2, 4, 6, 8])
```

Alternate elements were printed because the step-size was defined as 2. Notice that 10 was not printed as it was the last element.

Another similar function is **np.linspace()**, but instead of step size, it takes in the number of samples that need to be retrieved from the interval. A point to note here is that the last number is included in the values returned unlike in the case of np.arange().

```
np.linspace(0,1,5)

array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Great! Now you know how to create arrays using NumPy. But its also important to know the shape of the array.

The Shape and Reshaping of NumPy Arrays

Once you have created your ndarray, the next thing you would want to do is check the number of axes, shape, and the size of the ndarray.

Dimensions of NumPy arrays

You can easily determine the number of dimensions or axes of a NumPy array using the **ndims** attribute:

```
# number of axis
a = np.array([[5,10,15],[20,25,20]])
print('Array :','\n',a)
print('Dimensions :','\n',a.ndim)

Array :
 [[ 5 10 15]
 [20 25 20]]
```

```
Dimensions :
 2
```

This array has two dimensions: 2 rows and 3 columns.

Shape of NumPy array

The **shape** is an attribute of the NumPy array that shows how many rows of elements are there along each dimension. You can further index the shape so returned by the ndarray to get value along each dimension:

```
a = np.array([[1,2,3],[4,5,6]])
print('Array :','\n',a)
print('Shape :','\n',a.shape)
print('Rows = ',a.shape[0])
print('Columns = ',a.shape[1])

Array :
 [[1 2 3]
 [4 5 6]]
Shape :
 (2, 3)
Rows =  2
Columns =  3
```

Size of NumPy array

You can determine how many values there are in the array using the **size** attribute. It just multiplies the number of rows by the number of columns in the ndarray:

```
# size of array
a = np.array([[5,10,15],[20,25,20]])
print('Size of array :',a.size)
print('Manual determination of size of array :',a.shape[0]*a.shape[1])

Size of array : 6
Manual determination of size of array : 6
```

5	10	15
20	25	30

Shape : (2,3)
Size : 6
N-dim : 2

Reshaping a NumPy array

Reshaping a ndarray can be done using the **np.reshape()** method. It changes the shape of the ndarray without changing the data within the ndarray:

```
# reshape
a = np.array([3,6,9,12])
np.reshape(a,(2,2))

array([[ 3,  6],
       [ 9, 12]])
```

Here, I reshaped the ndarray from a 1-D to a 2-D ndarray.

While reshaping, if you are unsure about the shape of any of the axis, just input -1. NumPy automatically calculates the shape when it sees a -1:

```
a = np.array([3,6,9,12,18,24])
print('Three rows :','\n',np.reshape(a,(3,-1)))
print('Three columns :','\n',np.reshape(a,(-1,3)))

Three rows :
 [[ 3  6]
 [ 9 12]
 [18 24]]
Three columns :
 [[ 3  6  9]
 [12 18 24]]
```

Flattening a NumPy array

Sometimes when you have a multidimensional array and want to collapse it to a single-dimensional array, you can either use the **flatten()** method or the **ravel()** method:

```
a = np.ones((2,2))
b = a.flatten()
c = a.ravel()
print('Original shape :', a.shape)
print('Array :','\n', a)
print('Shape after flatten :',b.shape)
print('Array :','\n', b)
print('Shape after ravel :',c.shape)
print('Array :','\n', c)

Original shape : (2, 2)
Array :
 [[1. 1.]
 [1. 1.]]
Shape after flatten : (4,)
Array :
 [1. 1. 1. 1.]
Shape after ravel : (4,)
Array :
 [1. 1. 1. 1.]
```

But an important difference between flatten() and ravel() is that the former returns a copy of the original array while the latter returns a reference to the original array. This means any changes made to the array returned from ravel() will also be reflected in the original array while this will not be the case with flatten().

```
b[0] = 0
print(a)

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

The change made was not reflected in the original array.

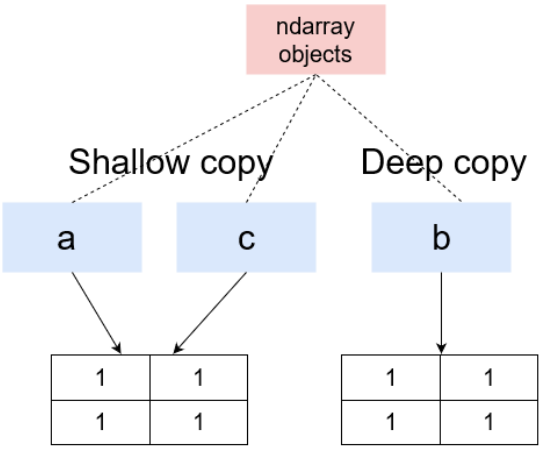
```
c[0] = 0
print(a)

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

But here, the changed value is also reflected in the original ndarray.

What is happening here is that flatten() creates a **Deep copy** of the ndarray while ravel() creates a **Shallow copy** of the ndarray.

Deep copy means that a completely new ndarray is created in memory and the ndarray object returned by flatten() is now pointing to this memory location. Therefore, any changes made here will not be reflected in the original ndarray.

A Shallow copy, on the other hand, returns a reference to the original memory location. Meaning the object returned by ravel() is pointing to the same memory location as the original ndarray object. So, definitely, any changes made to this ndarray will also be reflected in the original ndarray too.



Transpose of a NumPy array

Another very interesting reshaping method of NumPy is the **transpose()** method. It takes the input array and swaps the rows with the column values, and the column values with the values of the rows:

```
a = np.array([[1,2,3],
[4,5,6]])
b = np.transpose(a)
print('Original','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)

Original
 Shape (2, 3)
 [[1 2 3]
 [4 5 6]]
Expand along columns:
 Shape (3, 2)
 [[1 4]
 [2 5]
 [3 6]]
```

On transposing a 2 x 3 array, we got a 3 x 2 array. *Transpose has a lot of significance in linear algebra*.

Expanding and Squeezing a NumPy array

Expanding a NumPy array

You can add a new axis to an array using the expand_dims() method by providing the array and the axis along which to expand:

```
# expand dimensions
a = np.array([1,2,3])
b = np.expand_dims(a,axis=0)
c = np.expand_dims(a,axis=1)
print('Original:','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)
print('Expand along rows:','\n','Shape',c.shape,'\n',c)

Original:
 Shape (3,)
 [1 2 3]
Expand along columns:
 Shape (1, 3)
 [[1 2 3]]
Expand along rows:
 Shape (3, 1)
 [[1]
 [2]
 [3]]
```

Squeezing a NumPy array

On the other hand, if you instead want to reduce the axis of the array, use the **squeeze()** method. It removes the axis that has a single entry. This means if you have created a 2 x 2 x 1 matrix, squeeze() will remove the third dimension from the matrix:

```
# squeeze
a = np.array([[[1,2,3],
[4,5,6]]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Squeeze array:','\n','Shape',b.shape,'\n',b)

Original
 Shape (1, 2, 3)
 [[[1 2 3]
  [4 5 6]]]
Squeeze array:
 Shape (2, 3)
 [[1 2 3]
 [4 5 6]]
```

However, if you already had a 2 x 2 matrix, using squeeze() in that case would give you an error:

```
# squeeze
a = np.array([[1,2,3],
[4,5,6]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Squeeze array:','\n','Shape',b.shape,'\n',b)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-190-fcad729d251f> in <module>
      1 # squeeze
      2 a = np.array([[1,2,3],[4,5,6]])
----> 3 b = np.squeeze(a, axis=0)
      4 print('Original','\n','Shape',a.shape,'\n',a)
      5 print('Squeeze array:','\n','Shape',b.shape,'\n',b)

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

~\Anaconda3\lib\site-packages\numpy\core\fromnumeric.py in squeeze(a, axis)
   1436         return squeeze()
   1437     else:
-> 1438         return squeeze(axis=axis)
   1439
   1440

ValueError: cannot select an axis to squeeze out which has size not equal to one
```

Indexing and Slicing of NumPy array

So far, we have seen how to create a NumPy array and how to play around with its shape. In this section, we will see how to extract specific values from the array using indexing and slicing.

Slicing 1-D NumPy arrays

Slicing means retrieving elements from one index to another index. All we have to do is to pass the starting and ending point in the index like this: [start: end].

However, you can even take it up a notch by passing the step-size. What is that? Well, suppose you wanted to print every other element from the array, you would define your step-size as 2, meaning get the element 2 places away from the present index.

Incorporating all this into a single index would look something like this: [start:end:step-size].

```
a = np.array([1,2,3,4,5,6])
print(a[1:5:2])

[2 4]
```

Notice that the last element did not get considered. *This is because slicing includes the start index but excludes the end index.*

A way around this is to write the next higher index to the final index value you want to retrieve:

```
a = np.array([1,2,3,4,5,6])
print(a[1:6:2])

[2 4 6]
```

If you don’t specify the start or end index, it is taken as 0 or array size, respectively, as default. And the step-size by default is 1.

```
a = np.array([1,2,3,4,5,6])
print(a[:6:2])
print(a[1::2])
print(a[1:6:])

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

Slicing 2-D NumPy arrays

Now, a 2-D array has rows and columns so it can get a little tricky to slice 2-D arrays. But once you understand it, you can slice any dimension array!

Before learning how to slice a 2-D array, let's have a look at how to retrieve an element from a 2-D array:

```
a = np.array([[1,2,3],
[4,5,6]])
print(a[0,0])
print(a[1,2])
print(a[1,0])

1
6
4
```

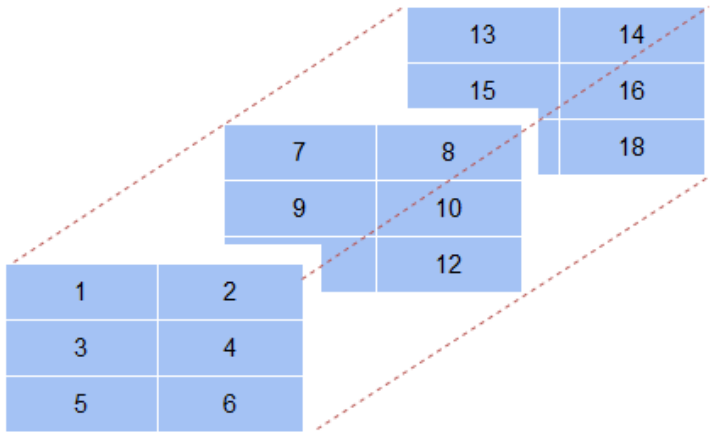
Here, we provided the row value and column value to identify the element we wanted to extract. While in a 1-D array, we were only providing the column value since there was only 1 row.

So, to slice a 2-D array, you need to mention the slices for both, the row and the column:

```
a = np.array([[1,2,3],[4,5,6]])
# print first row values
print('First row values :','\n',a[0:1,:])
# with step-size for columns
print('Alternate values from first row:','\n',a[0:1,::2])
#
print('Second column values :','\n',a[:,1::2])
print('Arbitrary values :','\n',a[0:1,1:3])

First row values :
 [[1 2 3]]
Alternate values from first row:
 [[1 3]]
Second column values :
 [[2]
 [5]]
Arbitrary values :
 [[2 3]]
```

Slicing 3-D NumPy arrays

So far we haven't seen a 3-D array. Let's first visualize how a 3-D array looks like:



```
a = np.array([[[1,2],[3,4],[5,6]],# first axis array
[[7,8],[9,10],[11,12]],# second axis array
[[13,14],[15,16],[17,18]]])# third axis array
# 3-D array
print(a)

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

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

 [[13 14]
  [15 16]
  [17 18]]]
```

In addition to the rows and columns, as in a 2-D array, a 3-D array also has a depth axis where it stacks one 2-D array behind the other. So, when you are slicing a 3-D array, you also need to mention which 2-D array you are slicing. This usually comes as the first value in the index:

```
# value
print('First array, first row, first column value :','\n',a[0,0,0])
print('First array last column :','\n',a[0,:,1])
print('First two rows for second and third arrays :','\n',a[1:,0:2,0:2])

First array, first row, first column value :
 1
First array last column :
 [2 4 6]
First two rows for second and third arrays :
 [[[ 7  8]
  [ 9 10]]

 [[13 14]
  [15 16]]]
```

If in case you wanted the values as a single dimension array, you can always use the flatten() method to do the job!

```
print('Printing as a single array :','\n',a[1:,0:2,0:2].flatten())

Printing as a single array :
 [ 7  8  9 10 13 14 15 16]
```

Negative slicing of NumPy arrays

An interesting way to slice your array is to use negative slicing. Negative slicing prints elements from the end rather than the beginning. Have a look below:

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

```
[ 5 10]
```

Here, the last values for each row were printed. If, however, we wanted to extract from the end, we would have to explicitly provide a negative step-size otherwise the result would be an empty list.

```
print(a[:,-1:-3:-1])

[[ 5  4]
 [10  9]]
```

Having said that, the basic logic of slicing remains the same, i.e. the end index is never included in the output.

An interesting use of negative slicing is to reverse the original array.

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Original array :','\n',a)
print('Reversed array :','\n',a[::-1,::-1])

Original array :
 [[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Reversed array :
 [[10  9  8  7  6]
 [ 5  4  3  2  1]]
```

You can also use the **flip()** method to reverse an ndarray.

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Original array :','\n',a)
print('Reversed array vertically :','\n',np.flip(a,axis=1))
print('Reversed array horizontally :','\n',np.flip(a,axis=0))

Original array :
 [[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Reversed array vertically :
 [[ 5  4  3  2  1]
 [10  9  8  7  6]]
Reversed array horizontally :
 [[ 6  7  8  9 10]
 [ 1  2  3  4  5]]
```

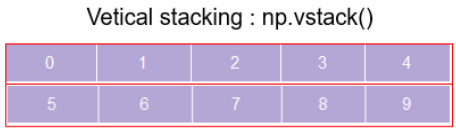
Stacking and Concatenating NumPy arrays

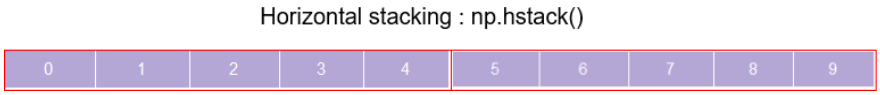
Stacking ndarrays

You can create a new array by combining existing arrays. This you can do in two ways:

- Either combine the arrays vertically (i.e. along the rows) using the **vstack()** method, thereby increasing the number of rows in the resulting array

- Or combine the arrays in a horizontal fashion (i.e. along the columns) using the **hstack()**, thereby increasing the number of columns in the resultant array






```
a = np.arange(0,5)
b = np.arange(5,10)
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Vertical stacking :','\n',np.vstack((a,b)))
print('Horizontal stacking :','\n',np.hstack((a,b)))

Array 1 :
 [0 1 2 3 4]
Array 2 :
 [5 6 7 8 9]
Vertical stacking :
 [[0 1 2 3 4]
 [5 6 7 8 9]]
Horizontal stacking :
 [0 1 2 3 4 5 6 7 8 9]
```

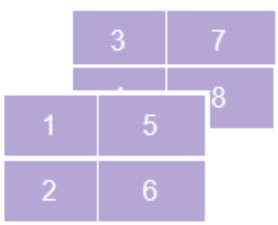
A point to note here is that the axis along which you are combining the array should have the same size otherwise you are bound to get an error!

```
a = np.arange(0,5)
b = np.arange(5,9)
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Vertical stacking :','\n',np.vstack((a,b)))
print('Horizontal stacking :','\n',np.hstack((a,b)))
```

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

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-127-b6a958ba57f9> in <module>
      3 print('Array 1 :','\n',a)
      4 print('Array 2 :','\n',b)
----> 5 print('Vertical stacking :','\n',np.vstack((a,b)))
      6 print('Horizontal stacking :','\n',np.hstack((a,b)))

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

~\Anaconda3\lib\site-packages\numpy\core\shape_base.py in vstack(tup)
    280     if not isinstance(arrs, list):
    281         arrs = [arrs]
--> 282     return _nx.concatenate(arrs, 0)
    283
    284

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

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimens
at index 0 has size 5 and the array at index 1 has size 4
```

Another interesting way to combine arrays is using the **dstack()** method. It combines array elements index by index and stacks them along the depth axis:

```
a = [[1,2],[3,4]]
b = [[5,6],[7,8]]
c = np.dstack((a,b))
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Dstack :','\n',c)
print(c.shape)

Array 1 :
 [[1, 2], [3, 4]]
Array 2 :
 [[5, 6], [7, 8]]
Dstack :
 [[[1 5]
  [2 6]]

 [[3 7]
  [4 8]]]
(2, 2, 2)
```



Concatenating ndarrays

While stacking arrays is one way of combining old arrays to get a new one, you could also use the **concatenate()** method where the passed arrays are joined along an existing axis:

```
a = np.arange(0,5).reshape(1,5)
b = np.arange(5,10).reshape(1,5)
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Concatenate along rows :','\n',np.concatenate((a,b),axis=0))
print('Concatenate along columns :','\n',np.concatenate((a,b),axis=1))

Array 1 :
 [[0 1 2 3 4]]
Array 2 :
 [[5 6 7 8 9]]
Concatenate along rows :
 [[0 1 2 3 4]
 [5 6 7 8 9]]
Concatenate along columns :
 [[0 1 2 3 4 5 6 7 8 9]]
```

The drawback of this method is that the original array must have the axis along which you want to combine. Otherwise, get ready to be greeted by an error.

Another very useful function is the **append** method that adds new elements to the end of a ndarray. This is obviously useful when you already have an existing ndarray but want to add new values to it.

```
# append values to ndarray
a = np.array([[1,2],
                [3,4]])
np.append(a,[[5,6]], axis=0)

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

Broadcasting in NumPy arrays – A class apart!

Broadcasting is one of the best features of ndarrays. It lets you perform arithmetics operations between ndarrays of different sizes or between an ndarray and a simple number!

Broadcasting essentially stretches the smaller ndarray so that it matches the shape of the larger ndarray:

```
a = np.arange(10,20,2)
b = np.array([[2],[2]])
print('Adding two different size arrays :','\n',a+b)
print('Multiplying an ndarray and a number :',a*2)

Adding two different size arrays :
 [[12 14 16 18 20]
 [12 14 16 18 20]]
Multiplying an ndarray and a number : [20 24 28 32 36]
```

Its working can be thought of like stretching or making copies of the scalar, the number, [2, 2, 2] to match the shape of the ndarray and then perform the operation element-wise. But no such copies are being made. It is just a way of thinking about how broadcasting is working.

This is very useful because it is more efficient to multiply an array with a scalar value rather than another array! It is important to note that two ndarrays can broadcast together only when they are compatible.

Ndarrays are compatible when:

1. Both have the same dimensions

2. Either of the ndarrays has a dimension of 1. The one having a dimension of 1 is broadcast to meet the size requirements of the larger ndarray

In case the arrays are not compatible, you will get a ValueError.

```
a = np.ones((3,3))
b = np.array([2])
a+b

array([[3., 3., 3.],
       [3., 3., 3.],
       [3., 3., 3.]])
```

Here, the second ndarray was stretched, hypothetically, to a 3 x 3 shape, and then the result was calculated.

NumPy Ufuncs – The secret of its success!

Python is a dynamically typed language. This means the data type of a variable does not need to be known at the time of the assignment. Python will automatically determine it at run-time. While this means a cleaner and easier code to write, it also makes Python sluggish.

This problem manifests itself when Python has to do many operations repeatedly, like the addition of two arrays. This is so because each time an operation needs to be performed, Python has to check the data type of the element. This problem is overcome by NumPy using the **ufuncs** function.

The way NumPy makes this work faster is by using **vectorization**. Vectorization performs the same operation on ndarray in an element-by-element fashion in a compiled code. So the data types of the elements do not need to be determined every time, thereby performing faster operations.

ufuncs are **Universal functions** in NumPy that are simply mathematical functions. They perform fast element-wise functions. They are called automatically when you are performing simple arithmetic operations on NumPy arrays because they act as wrappers for NumPy ufuncs.

For example, when adding two NumPy arrays using ‘+’, the NumPy ufunc add() is automatically called behind the scene and quietly does its magic:

```
a = [1,2,3,4,5]
b = [6,7,8,9,10]
%timeit a+b
```

```
283 ns ± 20.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
a = np.arange(1,6)
b = np.arange(6,11)
%timeit a+b
```

```
1.29 µs ± 48.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

You can see how the same addition of two arrays has been done in significantly less time with the help of NumPy ufuncs!

Maths with NumPy arrays

Here are some of the most important and useful operations that you will need to perform on your NumPy array.

Basic arithmetic operations on NumPy arrays

The basic arithmetic operations can easily be performed on NumPy arrays. The important thing to remember is that these simple arithmetics operation symbols just act as wrappers for NumPy ufuncs.

```
print('Subtract :',a-5)
print('Multiply :',a*5)
print('Divide :',a/5)
print('Power :',a**2)
print('Remainder :',a%5)

Subtract : [-4 -3 -2 -1 0]
Multiply : [ 5 10 15 20 25]
Divide : [0.2 0.4 0.6 0.8 1. ]
Power : [ 1 4 9 16 25]
Remainder : [1 2 3 4 0]
```

Mean, Median and Standard deviation

To find the mean and standard deviation of a NumPy array, use the **mean()**, **std()** and **median()** methods:

```
a = np.arange(5,15,2)
print('Mean :',np.mean(a))
print('Standard deviation :',np.std(a))
print('Median :',np.median(a))

Mean : 9.0
Standard deviation : 2.8284271247461903
Median : 9.0
```

Min-Max values and their indexes

Min and Max values in an ndarray can be easily found using the **min()** and **max()** methods:

```
a = np.array([[1,6],
[4,3]])
# minimum along a column
print('Min :',np.min(a,axis=0))
# maximum along a row
print('Max :',np.max(a,axis=1))

Min : [1 3]
Max : [6 4]
```

You can also easily determine the index of the minimum or maximum value in the ndarray along a particular axis using the **argmin()** and **argmax()** methods:

```
a = np.array([[1,6,5],
[4,3,7]])
# minimum along a column
print('Min :',np.argmin(a,axis=0))
# maximum along a row
print('Max :',np.argmax(a,axis=1))

Min : [0 1 0]
Max : [1 2]
```

Let me break down the output for you. The minimum value for the first column is the first element along the column. For the second column, it is the second element. And for the third column, it is the first element.

You can similarly determine what the output for maximum values indicates.

Sorting in NumPy arrays

For any programmer, the time complexity of any algorithm is of prime essence. Sorting is an important and very basic operation that you might well use on a daily basis as a data scientist. So, it is important to use a good sorting algorithm with minimum time complexity.

The NumPy library is a legend when it comes to sorting elements of an array. It has a range of sorting functions that you can use to sort your array elements. It has implemented quicksort, heapsort, mergesort, and timesort for you under the hood when you use the **sort()** method:

```
a = np.array([1,4,2,5,3,6,8,7,9])
np.sort(a, kind='quicksort')

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

You can even sort the array along any axis you desire:

```
a = np.array([[5,6,7,4],
                [9,2,3,7]])# sort along the column
print('Sort along column :','\n',np.sort(a, kind='mergresort',axis=1))
```

```
# sort along the row
print('Sort along row :','\n',np.sort(a, kind='mergresort',axis=0))

Sort along column :
 [[4 5 6 7]
 [2 3 7 9]]
Sort along row :
 [[5 2 3 4]
 [9 6 7 7]]
```

NumPy arrays and Images

NumPy arrays find wide use in storing and manipulating image data. But what is image data really?

Images are made up of pixels that are stored in the form of an array. Each pixel has a value ranging between 0 to 255 – 0 indicating a black pixel and 255 indicating a white pixel. A colored image consists of three 2-D arrays, one for each of the color channels: Red, Green, and Blue, placed back-to-back thus making a 3-D array. Each value in the array constitutes a pixel value. So, the size of the array depends on the number of pixels along each dimension.

Have a look at the image below:

Python can read the image as an array using the **scipy.misc.imread()** method in the SciPy library. And when we output it, it is simply a 3-D array containing the pixel values:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import misc

# read image
im = misc.imread('./original.jpg')
# image
im

array([[[115, 106,  67],
          [113, 104,  65],
          [112, 103,  64],
          ...,
          [160, 138,  37],
          [160, 138,  37],
          [160, 138,  37]],

         [[117, 108,  69],
          [115, 106,  67],
          [114, 105,  66],
          ...,
```

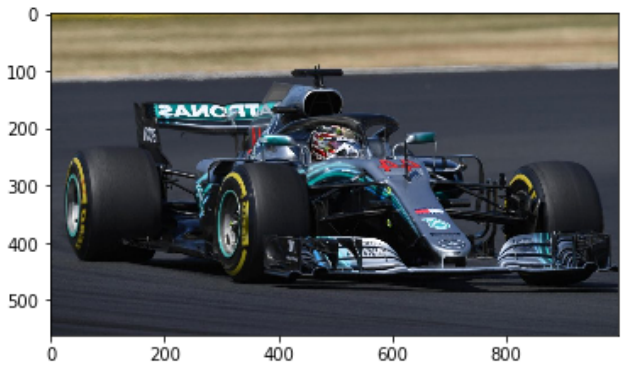
```
          [157, 135,  36],
          [157, 135,  34],
          [158, 136,  37]],

         [[120, 110,  74],
          [118, 108,  72],
          [117, 107,  71],
          ...,
```

We can check the shape and type of this NumPy array:

```
print(im.shape)
print(type(type))

(561, 997, 3)
numpy.ndarray
```

Now, since an image is just an array, we can easily manipulate it using an array function that we have looked at in the article. Like, we could flip the image horizontally using the **np.flip()** method:

```
# flip
plt.imshow(np.flip(im, axis=1))
```



Or you could normalize or change the range of values of the pixels. This is sometimes useful for faster computations.

```
im/255

array([[[0.45098039, 0.41568627, 0.2627451 ],
          [0.44313725, 0.40784314, 0.25490196],
          [0.43921569, 0.40392157, 0.25098039],
          ...,
          [0.62745098, 0.54117647, 0.14509804],
          [0.62745098, 0.54117647, 0.14509804],
          [0.62745098, 0.54117647, 0.14509804]],

         [[0.45882353, 0.42352941, 0.27058824],
          [0.45098039, 0.41568627, 0.2627451 ],
          [0.44705882, 0.41176471, 0.25882353],
          ...,
          [0.61568627, 0.52941176, 0.14117647],
          [0.61568627, 0.52941176, 0.13333333],
```

```
         [0.61960784, 0.53333333, 0.14509804]],

        [[0.47058824, 0.43137255, 0.29019608],
         [0.4627451 , 0.42352941, 0.28235294],
         [0.45882353, 0.41960784, 0.27843137],
         ...,
         [0.6       , 0.52156863, 0.14117647],
         [0.6       , 0.52156863, 0.13333333],
         [0.6       , 0.52156863, 0.14117647]],

        ...,
```