import numpy as np

- Numpy stands for Numerical Python
- It is a fundamental python package for scientific computing
- It can be used for arrays, Linear algebra, Random Numbers, Broadcasting

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

In [2]:
```
arr=np.array([1,2,3,4]) #print the array
arr/3 # divides all the elements in the arrays and prints float values
```

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

- Array: it is collection of elements of same data type
- Create arrays from lists or Tuples
- Arrays()

np.array(object, dtype, copy, order, subok, ndmin) #internal parameters

In [3]:
```
np.array([1,2,3]) #1d array
```

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

In [4]:
```
np.array([1,2,3.0]) # 1d array with different data types, also called upcasting
```

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

In [5]:
```
np.array([[1,2],[3,4]]) #2d array, nested list
```

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

In [6]:
```
np.array([1,2,3], ndmin=1) # prints 1d array without nested list
```

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

In [7]:
```
np.array([1,2,3], dtype=complex) #changes the data type, prints complex values
```

Out[7]: `array([1.+0.j, 2.+0.j, 3.+0.j])`

- Arange(): creates an array of evenly spaced values

np.arange([start,]stop[,step,]dtype=None) #includes start value but excludes stop value

In [8]:
```
np.arange(1,10) # prints array of 1-10, since we didn't gave step value it takes step as 1
```

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

In [9]:
```
np.arange(3) #we have given the stop value, so it starts from 0 by default
```

Out[9]: `array([0, 1, 2])`

In [10]:
```
np.arange(1,10,2) #starting value 1, stop value is 10 and step is 2
```

Out[10]: `array([1, 3, 5, 7, 9])`

In [11]:
```
np.arange(20,dtype="complex") #prints 0-20 complex numbers
```

Out[11]:
```
array([ 0.+0.j,  1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j,  5.+0.j,  6.+0.j,
        7.+0.j,  8.+0.j,  9.+0.j, 10.+0.j, 11.+0.j, 12.+0.j, 13.+0.j,
       14.+0.j, 15.+0.j, 16.+0.j, 17.+0.j, 18.+0.j, 19.+0.j])
```

- zeros(): Returns an array filled with zeros

np.zeros(shape, dtype=float, order= 'C')

In [12]:
```
np.zeros(5) #prints an array of 5 zeros of float data type
```

Out[12]: `array([0., 0., 0., 0., 0.])`

In [13]:
```
np.zeros(5, dtype="int") #prints an array of 5 zeros of integer data type
```

Out[13]: `array([0, 0, 0, 0, 0])`

In [14]:
```
np.zeros((3,5)) #we have given a tuple of int's,it prints the matrix of zeros with 3 rows 5 columns
```

Out[14]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [15]:
```
np.zeros([3,4]) #we have given list of int's,it prints the matrix of zeros with 3 rows, 4 columns
```

Out[15]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

*order doesn't effect the data, because it is for storing the multi dimensional data

- ones(): Creates an array filled with ones

np.ones(shape, dtype=None, order='C')

In [16]:
```
np.ones(8) #prints an array of 8 ones of float data type
```

Out[16]: `array([1., 1., 1., 1., 1., 1., 1., 1.])`

In [17]:
```
np.ones(5, dtype='int') #prints an array of 5 ones of integer data type
```

Out[17]: `array([1, 1, 1, 1, 1])`

In [18]:
```
np.ones([3,4]) #we have given list of integers, it prints the matrix of 1's with 3 rows and 4 columns
```

Out[18]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

- empty(): Creates an array without initializing the entries

np.empty(shape, dtype=float, order='C')

In [19]:
```
np.empty([3,4], dtype='int') #prints a matrix of values with 3 rows and 4 column of int data type
```

Out[19]:
```
[illegible]
```

- linspace(): Creates an array of filled evenly spaced values

np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

- geomspace() and logspace() will be the same thing but printing the values in geometric progressions

*num= evenly space values and the end point is optionally excluded.

In [20]:
```
#prints 5 values in between 2 to 3 which are evenly spaced and prints along with ending point also float data type
np.linspace(2, 3, num=5)
```

Out[20]: `array([2.  , 2.25, 2.5 , 2.75, 3.  ])`

In [21]:
```
#prints 5 values in between 2 to 3 which are evenly spaced and prints along without ending point as float data type
np.linspace(2, 3, num=5, endpoint=False)
```

Out[21]: `array([2. , 2.2, 2.4, 2.6, 2.8])`

In [22]:
```
#prints 5 values in between 2 to 3 which are evenly spaced and prints along with ending point and prints step value as well as float data type
np.linspace(2, 3, num=5,retstep=True)
```

Out[22]: `(array([2.  , 2.25, 2.5 , 2.75, 3.  ]), 0.25)`

- eye(): Returns array filled with zeros except in the kth diagonal, whose values are equal to 1

np.eye(N, M=None, k=0, dtype=<class 'float'>, order='C') # n= no.of rows, m= no.of columns

In [23]:
```
np.eye(5) #prints 1 in the main diagonal and remaining as zeros of float data type
```

Out[23]:
```
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 [24]:
```
np.eye(2,3) # prints 2 rows, 3 columns
```

Out[24]:
```
array([[1., 0., 0.],
       [0., 1., 0.]])
```

In [25]:
```
np.eye(4, k=-1) #main diagonal will be zero and 1's will be printed in below the main diagonal
```

Out[25]:
```
array([[0., 0., 0., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

- identity(): Returns the identity array

np.identity(shape, dtype = float)

In [26]:
```
np.identity(4) #prints the square identity matrix of float numbers
```

Out[26]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

random(): Random number generation

- np.rand(): uniformly distributed values
- np.randn(): Normally distributed values
- ranf(): uniformly distributed floating point numbers
- np.randint(): uniformly distributed integers in a given range

1) rand()

np.random.rand(d0,d1,d2,...) # prints the random values in a given shape.

In [27]:
```
np.random.rand(5) # prints an array of 5 random values
```

Out[27]: `[illegible]`

In [28]:
```
np.random.rand(3,5) # prints the matrix with 3 rows and 5 columns of random values
```

Out[28]:
```
array([[illegible],
       [0.45996382, 0.45714917, 0.61844475, 0.43612559, 0.05011898],
       [0.7165019 , 0.65428755, 0.26037702, 0.97425024, 0.82332967],
       [0.42025532, 0.38201385, 0.03044834, 0.13106546, 0.99632555]])
```

2) randn()

np.random.randn(d0,d1,d2,.....) # returns a sample from the standard normal distribution.

In [29]:
```
np.random.randn() # prints single random value
```

Out[29]: `0.18344853330952934`

In [30]:
```
np.random.randn(5) # prints 5 random values from the Standard normal distribution
```

Out[30]: `array([ 2.49096418, -0.48033878,  0.45218456,  0.47406426,  0.27937013])`

3) ranf()

np.random.ranf(size) # returns random floats in half open interval

In [31]:
```
np.random.ranf(5) # prints 5 random float values
```

Out[31]: `array([0.52029337, 0.1037143 , 0.26668696, 0.8373044 , 0.29213447])`

In [32]:
```
5*np.random.random_sample(3)
```

Out[32]: `array([0.79983065, 2.56311646, 4.94827553])`

4)randint()

np.random.randint(low, high=None, size=None, dtype= '1')# returns a random integer from low to high in half open interval.

In [33]:
```
#2 is assigned to low, generates 10 random values, we get 0,1 because high value is not mentioned
np.random.randint(1, size =10)
```

Out[33]: `array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])`

In [34]:
```
np.random.randint(5, size =(2,4)) # prints random values 2d array with 2 rows and 4 columns
```

Out[34]:
```
array([[4, 4, 1, 3],
       [1, 3, 0, 3]])
```

- Attributes of arrays
  - Dimension: N-dimensional array, if you want to know the dimension of an array We can use 'ndim' attribute

In [35]:
```
a=np.array([1,2,3,4])
a.ndim #prints the dimension of array.
```

Out[35]: `1`

- Shape: tuple of elements indicating the number of elements that are stored along each dimension of the array.

In [36]:
```
a=np.zeros(5)
a.shape #prints 5 number of elements in array
```

Out[36]: `(5,)`

In [37]:
```
b= np.array([[1,2],[3,4]])
b.shape # prints the number of rows and number of columns (2,2)
```

Out[37]: `(2, 2)`

- Size: This will tell the total number of elements in the array.

In [38]:
```
b= np.array([[1,2],[3,4]])
b.size #prints the number of elements in the 2 d array (4)
```

Out[38]: `4`

- Dtype: Describes the data types
- Itemsize: the size of each element in bytes.

INDEXING

- We can access the element of array through index, we can access single value of array through index
- Array follows Zero based indexing i.e, first element index is zero. We can also use negative indexing for arrays

In [39]:
```
a=np.array([1,2,3,4,5]) #1D array with list
```

In [40]:
```
a[3] # prints 4
```

Out[40]: `4`

In [41]:
```
a[-2] #prints 4 again
```

Out[41]: `4`

In [42]:
```
b=np.array([[1,2],[3,4]]) #2D array with list
b[0][0] # zero row , zero column , prints 1
```

Out[42]: `1`

In [43]:
```
b[1] #prints row ([3,4])
```

Out[43]: `array([3, 4])`

In [44]:
```
c=np.array([[[1,2,3,4], [5,6,7,8], [9,10,11,12]], [[13,14,15,16], [17,18,19,20], [21,22,23,24]]])
c[0][0][1] #zero i, zero j, 1 k, prints 2
```

Out[44]: `2`

In [45]:
```
c[-2][-3][-3] # prints -2
```

Out[45]: `2`

SLICING

- To retrieve a collection of arrays, we use slicing.

a[start : stop: step]

In [46]:
```
a=np.array([1,2,3,4])
a[1:4] # prints ([2,3,4]) , step=1 by default
```

Out[46]: `array([2, 3, 4])`

In [47]:
```
a[1::2] #prints ([2,4]), it prints every second element from starting point
```

Out[47]: `array([2, 4])`

In [48]:
```
a[:] #prints entire array
```

Out[48]: `array([1, 2, 3, 4])`

b[start : end: step, start:end:step]

In [49]:
```
b=np.array([[1,2],[2,3],[3,4]])
b[1:,1:] #prints ([[3],[4]])
```

Out[49]:
```
array([[3],
       [4]])
```

c[start : end: step, start : end : step, start : end : step] #number of matrices, rows, columns

In [50]:
```
c=np.array([[[1,2,3,4], [5,6,7,8], [9,10,11,12]], [[13,14,15,16], [17,18,19,20], [21,22,23,24]]])
c[:,:,1] #prints ([[[1],[5],[9]],[[13],[17][21]])
```

Out[50]:
```
array([[ 2,  6, 10],
       [14, 18, 22]])
```

ARITHMETIC OPERATION

- We can do addition, subtraction, multiplication, division on arrays
- Arithmetic operations on arrays are applied element by element in arrays

In [51]:
```
a=np.array([1,2,3,4])
a+2 #add 2 to the each element of array ([3,4,5,6])
```

Out[51]: `array([3, 4, 5, 6])`

- Similarly subtraction, multiplication, division

In [52]:
```
b=np.array([6,7,8,9])
a+b #prints sum of two array ([7,9,11,13])
```

Out[52]: `array([ 7,  9, 11, 13])`

- Similarly subtraction, multiplication, division
- Similarly for 2-D arrays of same shape as well

1) a+b = np.add(a,b)
2) a-b = np.subtract(a,b)
3) a*b = np.multiply(a,b)
4) a/b = np.divide(a,b)
5) a%b = np.mod(a,b)
6) a**b = np.power(a,b)

BROADCASTING

- Can we perform arithmetic operations on arrays of different shape and size.
- Broad Casting allows us to perform arithmetic operations on arrays of different size or shape.
- Broadcasting will stretch the value/array to0 the required shape, the performs arithmetic operation

- BroadCasting Rules
  - The size of each dimension should be same
  - The size of one of the dimension should be one

1) If the two arrays differ in their number of dimensions, the shape of the one earth
fewer dimension is padded with ones on its leading side(left side)
2) If the shape of the two arrays does not match in any dimensions, the array with
shape equal to 1 in that dimension is stretched to match the other shape.
3) If in any dimension the size disagrees and neither equal to 1, an error is raised.

In [53]:
```
# This code gives Broad casting error because of their shapes
a=np.array([10,20,30])
b=np.array([1,2,3,4])
a+b # raises error due to broadcasting error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-53-c3d810cdbfa5> in <module>
      1 a=np.array([10,20,30])
      2 b=np.array([1,2,3,4])
----> 3 a+b # raises error due to broadcasting error

ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

In [54]:
```
a=np.array([[1,2],[3,4],[5,6]])
b=np.array([10,20])
a.shape #prints(3,2)
```

Out[54]: `(3, 2)`

In [56]:
```
b.shape #prints(2,)
```

Out[56]: `(2,)`

In [57]:
```
a+b #prints suum of the arrays
```

Out[57]:
```
array([[11, 22],
       [13, 24],
       [15, 26]])
```

Because,
1) Rule 1: According to Rule 1, if there are two different dimensions, we need to add 1 for the b array.
a=(3,2) b=(1,2)
2) Rule 2: Right values of both the arrays were matched(2=2), (3!=1)
3) Rule 3:According to rule 3, any one of the dimension should be 1, in b array it is (1,2) it satisfied
4) Hence, we will get output in higher dimensions.

ARRAY MANIPULATION

- reshape(): Gives the new shape to the array without changing the array data .

np.reshape(array, shape, order='C')

In [58]:
```
a=np.arange(10) #prints array of 0-10
```

In [59]:
```
b=np.reshape(a,(5,2)) # changes 1D array to 2D array without changing data (rowwise )
```

In [61]:
```
b=np.reshape(a,(5,2), order='F') # changes 1D array to 2D array without changing data (column wise
```

- Size of the array is nothing but product of the shape
- When we change the shape the size should remain the same, that shouldn't change.

- resize(): This array will change the data of array
- If the size of new array is larger than the original array, then it makes the repeated copies of data

np.resize(array, shape)

In [63]:
```
a = np.arange(5) #creates a 1d array from 0 to 5
np.resize(a,(2,3)) #changes the shape and size of the array without
```

Out[63]:
```
array([[0, 1, 2],
       [3, 4, 0]])
```

FLATTEN

- flatten(): Returns the copy of array which collapsed into 1 Dimension
- If we give 2d or 3d array it makes a copy of array collapsed into 1d

array.flatten(order= 'C')

In [65]:
```
a=np.array([[[1,2,3],[3,4,5]], [[6,7,8],[9,10,11]]])
a.flatten()
```

Out[65]: `array([ 1,  2,  3,  3,  4,  5,  6,  7,  8,  9, 10, 11])`

- This gives the copy of array in 1D, it converts 3D to 1D in row wise

In [67]:
```
a.flatten(order= 'F')
```

Out[67]: `array([ 1,  6,  3,  9,  2,  7,  4, 10,  3,  8,  5, 11])`

- This gives the copy of array in 1D, columns wise
- ravel(): Also used to flatten the array, but copy is made only if needed

np.ravel(array, order= 'C')

In [68]:
```
a.ravel() # This also flatten the given array.
```

Out[68]: `array([ 1,  2,  3,  3,  4,  5,  6,  7,  8,  9, 10, 11])`

TRANSPOSE AND SWAPAXES

- transpose(): Rearrange the dimension of the array

np.transpose(array, axes=None)

In [69]:
```
a = np.arange(1,11).reshape(5,2) #prints the matrix with 5 rows and 2 columns
np.transpose(a) # transposes the matrix into 2 row and 5 columns
```

Out[69]:
```
array([[ 1,  3,  5,  7,  9],
       [ 2,  4,  6,  8, 10]])
```

- It just reverses the shape of the array or matrix
- If we apply transpose on 1D array we get the same array

In [70]:
```
a.T # this also transposes the matrix
```

Out[70]:
```
array([[ 1,  3,  5,  7,  9],
       [ 2,  4,  6,  8, 10]])
```

- swapaxes(): This is used to interchange the any two axes of a given array.

np.swapaxes(array, axis1,axis2)

In [73]:
```
a=np.array([[1,2],[3,4]])
np.swapaxes(a,0,1) #swaps 1st column into 1st row, 2nd column into 2nd row
```

Out[73]:
```
array([[1, 3],
       [2, 4]])
```

- This will be useful when we are dealing with 3D or 4D array, when we want to change the two columns

In [ ]: