# Numpy

Installation of numpy. If you have Python and PIP already installed on a system, then installation of NumPy is very easy. Install it using this command. pip install numpy

In [1]:
```
pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\saich\anaconda3\lib\site-packages (1.26.4)
Note: you may need to restart the kernel to use updated packages.
```

Here I already installed so i am getting note as Requirement already satisfied.

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc

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

Now numpy is imported and ready to use

In [2]:
```
import numpy as np
arr=np.array([1,2,3,4])
print(arr)
```

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

Here np is the alias(alternate name) of numpy it is hard to call all the as numpy so we can assing a alias.Alias is upto us we can give any thing.

Cheacking type and version of numpy

In [4]:
```
print(type(arr))
print(np.__version__)
```

```
<class 'numpy.ndarray'>
1.26.4
```

Creating a Numpy ndarray Numpy is used work with arrays.The array object in numpy is called as ndarrar(n dimensional array) We can create a Numpy ndarray object by using the array() function

To create an ndarray, we can pass a list ,tuple or any array-like object into the array() method, and it will be converted into an ndarray.

In [8]:
```
a=(1,2,3)
arr=np.array((1,2,3))
print(arr)
print(type(a))
print(type(arr))
```

```
[1 2 3]
<class 'tuple'>
<class 'numpy.ndarray'>
```

Here we given a tuple but it is converted into array.

Dimensions in array. A dimension in arrays is one level of array depth(nested array) nested array: are array that have arrays as their elements

0-D arrays 0-D arrays,or scalars are the elementss in an array.Each value in an array is a 0-D array

Dimensions : Numpy provides the "ndim" attribute that returns an integer that tells us how many dimensions that the array have.

In [38]:
```
arrzd=np.array(89)
print(arrzd)
print("no.of dimensions is : " ,np.ndim(arrzd)) #no.of dimensions
```

```
89
no.of dimensions is :  0
```

1-D array An array that has 0-D array as its element is called as uni-dimensional or 1-D array . These are the most common and basic arrays.

In [37]:
```
arrod = np.array([89,99])
print(arrod)
print("no.of dimensions is : " ,np.ndim(arrod)) #no.of dimensions
```

```
[89 99]
no.of dimensions is :  1
```

2-D Arrays An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order

tensor

In [36]:
```
arrtd = np.array([[1,2,3,4],[8,7,6,5]])
print(arrtd)
print("no.of dimensions is : " ,np.ndim(arrtd)) #no.of dimensions
```

```
[[1 2 3 4]
 [8 7 6 5]]
no.of dimensions is :  2
```

3-D arrays An array that has 2-D arrays(matrices) as its elements is called as 3-D array. These are often used to represent a 3rd order tensor

In [35]:
```
arythd = np.array([[[1,2,3],[6,5,4]],[[9,8,7],[10,11,12]]])
print(arythd)
print("no.of dimensions is : " ,np.ndim(arythd)) #no.of dimensions
```

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

 [[ 9  8  7]
  [10 11 12]]]
no.of dimensions is :  3
```

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using th"e ndm"in argument.

In [42]:
```
ary = np.array([1,2,3,4],ndmin =5)
print(ary)
print("no.of dimensions : ",ary.ndim)
```

```
[[[[[1 2 3 4]]]]]
no.of dimensions :  5
```

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

## Accessing elements with index

Numpy array indexing Like as list here in numpy we access the elements by using their index. We can access an array element by referring to its index number. Like as noraml index here the index start with 0 and the second has index 1 etc. Depends on range we assign.

In [47]:
```
arr=np.array([1,2,3,4])
print(arr)
print(arr[0])
print(arr[2])
```

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

For getting selective elements we can use + .

In [52]:
```
arr = np.array([1,2,3,4,5,6,7,8])
print(arr)
print(arr[2],arr[4],arr[7])
print("addition of 2nd,4th and 7th elements : " ,arr[2] + arr[4] + arr[7])
```

```
[1 2 3 4 5 6 7 8]
3 5 8
addition of 2nd,4th and 7th elements :  16
```

Till now we access only one dimensional array now we are going to see multi-dimensional arrays.

In [3]:
```
arr = np.array([[1,2,3,4],[8,7,6,5]])
print(arr)
print(arr[0,2] , arr[0,1] , arr[1,2] , arr[1,0])
print("subration of 1st row 3rd and 2nd and 2nd row 3rd and 1st elements :",
      arr[0,2] - arr[0,1] - arr[1,2] - arr[1,0])
```

```
[[1 2 3 4]
 [8 7 6 5]]
3 2 6 8
subration of 1st row 3rd and 2nd and 2nd row 3rd and 1st elements : -13
```

In [61]:
```
arr = np.array([[[1,2,3,4],[5,6,7,8]],[[9,10,11,12],[13,14,15.16]]])
print(arr)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[61], line 1
----> 1 arr = np.array([[[1,2,3,4],[5,6,7,8]],[[9,10,11,12],[13,14,15.16]]])
      2 print(arr)

ValueError: setting an array element with a sequence. The requested array has an inhomogeneous shape after 2 dim
ensions. The detected shape was (2, 2) + inhomogeneous part.
```

Here we are getting error because the elements in each array has to be same length.

In [69]:
```
arr = np.array([[[1,2,3,4],[5,6,7,8]],[[9,10,11,12],[13,14,15,16]]])
print(arr)
print(arr[0,0,2])
print(arr[1,1,1])
```

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

 [[ 9 10 11 12]
  [13 14 15 16]]]
3
14
```

arr[0,0,2] prints the value 3 . Because,the first number represents the first dimension which contains two arrays [1,2,3,4], , , , 8 since we selected 0,we are left with first array: The second number represents the array in the dimension that is [1,2,3,4] and finally third number represents the index of the selected array i.e, 3.]

## Negative indexing

Using negative index also we can access the arrays

In [3]:
```
arr = np.array([[1,2,3,4],[5,6,7,8]])
print("Last element from 2nd dim : ",arr[-1,-1])
```

```
Last element from 2nd dim :  8
```

## Slicing arrays

Slicing in python means elements from one given index to another given index. We pass slice insted 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 as starting . If we din't pass end its considered length of array in that dimension. If we don't pass pass step its considered 1 as step value.

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

```
[2 3 4]
```

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

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

Note : The result of first argument includes the starting index , but excludes the end index.So the end will considered as length of array in that dimension.

Note : The result of second argument includes the ending index , but excludes the start index .So the start will considered as 0 index of array in that dimension.

Negative slicing Use negative index for negative slicing.But here index refer from end of the array.

In [10]:
```
arr = np.array([1,2,3,4,5,6,7,8])
print(arr[-5:-2])
print(arr[-2:-5]) #This is wrong way for negative index slicing we get a empty array
```

```
[4 5 6]
[]
```

Note : The starting negative index for slicing is start's from right side index of the array.If the slicing is done in [-2:-5] we will get a empty array

Step Use the step value to determine the step of the slicing As [start:end:step]

In [14]:
```
arr = np.array([1,2,3,4,5,6,7,8])
print(arr[1:6:2])
```

```
[2 4 6]
```

Here return elements from index 1 to index 6 with 2 steps

In [16]:
```
arr = np.array([1,2,3,4,5,6,7,8])
print(arr[::2])
```

```
[1 3 5 7]
```

Here the whole array is returning with step 2

2-D slicing

In [17]:
```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1,1:3])
```

```
[7 8]
```

Here returned elements from second array because we given [1,1:3] as input here first 1 act as the selection of array and the [1:3] act as the slicing the selected array

In [18]:
```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2,1:3])
```

```
[[2 3]
 [7 8]]
```

Here return the elements from two arrays as we ask to print.

Data Types in Python By default python have these data types: 1.string 2. integer 3.float 4.boolean 5.complex Like as pyhton Numpy also have its own data types they are: Data types in numpy: 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 typ ( void )

This how we cheacks the data tyoe in Numpy. In normal python we check data type by using key word type but in numpy we use varible_name.dtype

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

```
[1 2 3 4 5 6]
int32
```

In above output 32 is the number of bits

In [7]:
```
arstr = np.array(['python',"Numpy","Data science"])
print(arstr)
print(arstr.dtype)
```

```
['python' 'Numpy' 'Data science']
<U12
```

In normal python it is string in list data type.But in here in numpy it is u(uni code) with 12 bits.

We can assing a type to a array.

In [14]:
```
arr = np.array([1,2,3,4,5], dtype = 'S')
print(arr)
print(arr.dtype)
```

```
[b'1' b'2' b'3' b'4' b'5']
|S1
```

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

In [30]:
```
arr = np.array([1,2,3,7,6] , dtype = 'i4')
print(arr)
print(arr.dtype)
```

```
[1 2 3 7 6]
int32
```

In [33]:
```
arr = np.array(['a','2','3'],dtype = 'i')
print(arr)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[33], line 1
----> 1 arr = np.array(['a','2','3'],dtype = 'i')
      2 print(arr)

ValueError: invalid literal for int() with base 10: 'a'
```

Error for above code is because non integer string like "a" can not be converted to integer. For that we have option that astype() which

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

In [37]:
```
arr = np.array([1.2,3.1,9.4])
narr = arr.astype('i')
print(narr)
print(narr.dtype)
```

```
[1 3 9]
int32
```

In above code the decimal number is converted into integer as their round value.

In [40]:
```
arr = np.array([1,0.1,4,0,-3])
nary = arr.astype(bool)
print(nary)
print(nary.dtype)
```

```
[ True  True  True False  True]
bool
```

Nmmpy array copy and view. The difference between copy and view. The main difference between a copy and 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 in copy will not affect the original array and any changes made in orginal will not affect the copy array. The view does not own the data and any changes made in view will affect the original and if any changes made in orginal array will affect view.

Here we are copying and making changes in orginal array and printing both.

In [2]:
```
arr  = np.array([1,2,3,4,5,6])
a = arr.copy()
arr[0] = 87
print(arr)
print(a)
```

```
[87  2  3  4  5  6]
[1 2 3 4 5 6]
```

See the orginal array was changed and the copied array did't change.

Now here creating a view of orginal and making changes in orginal.

In [3]:
```
b = arr.view()
arr[2] = 45
print(arr)
print(b)
```

```
[87  2 45  4  5  6]
[87  2 45  4  5  6]
```

In [4]:
```
b[3]=78
print(arr)
print(b)
```

```
[87  2 45 78  5  6]
[87  2 45 78  5  6]
```

See the orginal and view both affects the changes.

As mentioned above ,copies owns the data ,and views doesn't. In numpy we have attrubute that checks whether it owns data or not if it has own data returns None otherwise ,the base attribute refers to the orginal object.

In [7]:
```
arr  = np.array([1,2,3,4,5,6])
a = arr.copy()
b = arr.view()
print(a.base)
print(b.base)
```

```
None
[1 2 3 4 5 6]
```

In python for finding the number of elements we have len attribute like that here we have shape. Shape of an array : The shape of an array is the number of elements in each dimension. For finding the number of elements in array we have a attribute shape.

In [10]:
```
arr = np.array([[1,2,3,4],[64,3,56,7]])
print(arr)
print(arr.shape)
```

```
[[ 1  2  3  4]
 [64  3 56  7]]
(2, 4)
```

In above output (2,4) 2 represents number of dimensions and 4 represents the number elements in each list.

In [11]:
```
arr = np.array([1,2,3,4,5], ndmin = 5)
print(arr)
print("shape of array : ",arr.shape)
```

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

See here the shape of array like (1,1,1,1,5) because all four dimension have one elements and only last fifth dimension has five values.

We can reshape the arrays on our requirements. Reshaping means changing the shape of an array. We know that shape means number of elements in each array. By reshaping we can add or remove dimensions or change number of elements in each dimension. For reshaping we use reshape attribute. For reshaping the should be in multiples of dimension we are giving i.e, for (4,2) reshaping the elements should be 8.

In [17]:
```
arr = np.array([0,1,2,3,5,6,7,8])
narr = arr.reshape(2,4)
print(arr)
print(narr)
```

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

In [38]:
```
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
narr = arr.reshape(2,3,2)
print(narr)
print(narr.ndim)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
3
```

In numpy we have option for giving unknow dimension. That means you do not have to specify an exact number for one of the dimensions in the resgaped method.Pass -1 as the value , and numpy will calculate this number for you

In [7]:
```
arr = np.array([0,1,2,3,5,6,7,8])
narr = arr.reshape(2,-1)
print(narr)
```

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

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

Till now we work with reshaping the single dimension into different dimension. So now we are going to work on changing different dimensions into single dimension.By just repalcing it as reshape(-1).

In [50]:
```
arr = np.array([[0,1,2,3],[5,6,7,8]])
narr = arr.reshape(-1)
print(narr)
```

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

In [45]:
```
arr = np.array([[0,1,2,3],[5,6,7,8]])
narr = arr.flatten()
print(narr)
```

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

Note: 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. These fall under Intermediate to Advanced section of numpy

Iterating arrays : iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy , we can do this using basic for loop of python. if we iterate on a 1-D array it will go through each element one by one.

Iterate on the elements of 1-D array

In [2]:
```
arr = np.array([1,2,3,4,5])
for i in arr:
    print(i)
```

```
1
2
3
4
5
```

Iterate on the elements of 2-D array.

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

for i in arr:
    print(i)
```

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

See here iterates through dimensionaly but when we deal with 1-D array it iterate through elements.

But,if we iterate on n-D array it will go through n-1th dimension one by one.

To get the actual values , the scalars ,we have to iterate the arrays in each dimension.

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

for i in arr:
    for j in i:
        print(j)
```

```
1
2
3
4
5
6
```

Here it iterates through elements.

Iterates through 3-D array.

In [13]:
```
arr = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
for i in arr:
    print("i represents the 2-D array")
    print(i)
```

```
i represents the 2-D array
[[1 2 3]
 [4 5 6]]
i represents the 2-D array
[[ 7  8  9]
 [10 11 12]]
```

In [14]:
```
arr = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
for i in arr:
    for j in i:
        for k in j:
            print(k)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

Iterating arrays using nditer(). 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. In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very high dimensionality.

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

```
1
2
3
4
5
6
7
8
9
10
11
12
```

Here we just call for loop once in our code but in normal we have to call 3 time .So by this we can minimize time and space complexty.

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 [7]:
```
arr = np.array([[1,2,3,4],[5,6,7,8]])
for i in np.nditer(arr,flags = ['buffered'],op_dtypes = ['S']):
    print(i)
```

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

Iterating with different step size.

In [11]:
```
arr = np.array([[1,2,3,4],[5,6,7,8]])
for i in np.nditer(arr[:,::2]):
    print(i)
```

```
1
3
5
7
```

## Enumerated iteration using ndenumerate()

Enumeration means mentioning sequence number of something one by one. Sometimes we requires corresponding index of the elements while iterating , the ndnumerate() method can be used for those usecases.

In [9]:
```
arr = np.array([1,2,3])
for i,j in np.ndenumerate(arr):
    print(i,j)
```

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

For 2-D array.

In [16]:
```
arr = np.array([[1,2,3,4,5],[6,7,8,9,10]])
for i,j in np.ndenumerate(arr):
    print(i,j)
```

```
(0, 0) 1
(0, 1) 2
(0, 2) 3
(0, 3) 4
(0, 4) 5
(1, 0) 6
(1, 1) 7
(1, 2) 8
(1, 3) 9
(1, 4) 10
```

Joining array. Joining means putting contents of two or more arrays in a single array. In SQL we join tables based on key (primary or foregin) , 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 axis .If axis is not explicitly passed , it taken as 0.

In [17]:
```
arr1 = np.array([1,2,3])
arr2 = np.array([6,5,4])
arr = np.concatenate((arr1,arr2))
```

```
print(arr)
```

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

Above we done only on 1-D arrays here we see in different different dimensions.

In [21]:
```
arr1 = np.array([[1,2,3],[4,5,6]])
arr2 = np.array([[7,8,9],[10,11,12]])
arr = np.concatenate((arr1,arr2))
print(arr)
print("Number of dimensions are : ",arr.ndim)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
Number of dimensions are :  2
```

Till now we didn't mention axis here we mention axis.

In [25]:
```
arr1 = np.array([[1,2,3],[4,5,6]])
arr2 = np.array([[7,8,9],[10,11,12]])
arr = np.concatenate((arr1,arr2),axis = 1)
print(arr)
```

```
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

For joining arrays not only concatenate we can use stack also. 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 [32]:
```
arr1 = np.array([1,2,3])
arr2 = np.array([6,5,4])
arr = np.stack((arr1,arr2))
print(arr)
```

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

In [33]:
```
arr1 = np.array([[1,2,3],[4,5,6]])
arr2 = np.array([[7,8,9],[10,11,12]])
arr = np.stack((arr1,arr2),axis = 1)
print(arr)
```

```
[[[ 1  2  3]
  [ 7  8  9]]

 [[ 4  5  6]
  [10 11 12]]]
```

Stacking along Rows. For this Numpy provides a helper function : hstack() to stack along rows

In [39]:
```
arr1 = np.array([1,2,3])
arr2 = np.array([6,5,4])
arr = np.hstack((arr1,arr2))
print(arr)
```

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

In [35]:
```
arr1 = np.array([[1,2,3],[4,5,6]])
arr2 = np.array([[7,8,9],[10,11,12]])
arr = np.hstack((arr1,arr2))
print(arr)
```

```
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

Stacking along columns. For stacking along column Numpy provides : vstack()

In [40]:
```
arr1 = np.array([1,2,3])
arr2 = np.array([6,5,4])
arr = np.vstack((arr1,arr2))
print(arr)
```

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

In [43]:
```
arr1 = np.array([[1,2,3],[4,5,6]])
arr2 = np.array([[7,8,9],[10,11,12]])
arr = np.vstack((arr1,arr2))
print(arr)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Stacking along height (depth): For stacking in depth Numpy provides function : dstack()

In [49]:
```
arr1 = np.array([1,2,3])
arr2 = np.array([6,5,4])
arr = np.dstack((arr1,arr2))
print(arr)
```

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

## Numpy Splitting array.

Splitting is nothing but, reverse operation of Joining. Joining merges multiple arrays into one and Splitting breaks one array into multiple. We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits

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

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

Note: The return value is a list containing two arrays. If the array has less elements than required, it will adjust from the end accordingly.

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

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

Note: 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 [13]:
```
arr = np.array([1,2,3,4,5,6])
new_arr = np.split(arr,3)
print(new_arr)
```

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

In [14]:
```
arr = np.array([1,2,3,4,5,6])
new_arr = np.split(arr,4)
print(new_arr)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[14], line 2
      1 arr = np.array([1,2,3,4,5,6])
----> 2 new_arr = np.split(arr,4)
      3 print(new_arr)

File ~\anaconda3\Lib\site-packages\numpy\lib\shape_base.py:864, in split(ary, indices_or_sections, axis)
    862     N = ary.shape[axis]
    863     if N % sections:
--> 864         raise ValueError(
    865             'array split does not result in an equal division') from None
    866 return array_split(ary, indices_or_sections, axis)

ValueError: array split does not result in an equal division
```

See here we are getting error because we given shape as 4 so it desn't adjust the elements

Split into arrays The return value of the array_split() method is an array containing each of the split as an array. If you split an array into 3 arrays , you can access them from the result just like any array elements.

In [6]:
```
arr = np.array([1,2,3,4,5,6])
new_arr = np.array_split(arr,3)
print(new_arr[0])
print(new_arr[1])
print(new_arr[2])
```

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

## Splitting 2-D arrays

Use the same syntax when aplitting 2-D arrays. Use the array_split() method , pass in the array you want to split and the number of split you want to do.

In [12]:
```
arr = np.array([[1,2,3],[7,4,6,],[5,4,6,],[6,7,8]])
new_arr = np.array_split(arr,4)
print(new_arr)
```

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

The above code return four 2-D arrays. In addition , you can specify which axis you want to do the split around.

In [17]:
```
arr = np.array([[1,2,3],[7,4,6,],[5,4,6,],[6,7,8]])
new_arr = np.array_split(arr,4,axis = 1)
print(new_arr)
```

```
[array([[1],
       [7],
       [5],
       [6]]), array([[2],
       [4],
       [4],
       [7]]), array([[3],
       [6],
       [6],
       [8]]), array([], shape=(4, 0), dtype=int32)]
```

We can use hsplit( ) insted of array_split( ) like as hstack( )

In [15]:
```
arr = np.array([[1,2,3],[7,4,6,],[5,4,6,],[6,7,8]])
new_arr = np.hsplit(arr,3)
print(new_arr)
```

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

Note: Similar alternates to vstack() and dstack() are available as vsplit() and dsplit()

In [22]:
```
arr = np.array([[1,2,3],[7,4,6,],[5,4,6,],[6,7,8]])
new_arr = np.vsplit(arr,4)
print(new_arr)
```

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

In [29]:
```
arr = np.array([[[1,2,3],[7,4,6,]],[[5,4,6,],[6,7,8]]])
new_arr = np.dsplit(arr,3)
print(new_arr)
```

```
[array([[[1],
        [7]],

       [[5],
        [6]]]), array([[[2],
        [4]],

       [[4],
        [7]]]), array([[[3],
        [6]],

       [[6],
        [8]]])]
```

## Numpy Search Array

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

In [34]:
```
arr = np.array([1,2,3,4,5,6,4,7,8,9,10,3])
se = np.where( arr == 3)
print(se)
```

```
(array([ 2, 11], dtype=int64),)
```

In above code output is a tuple with (array [2,11] , dtype = int64) first [2,11] is the index of the searched 3 and the second is the type of the searched element. Which means that the value 3 is present at index 2,11.

In [35]:
```
arr = np.array([1,2,3,4,5,6,4,7,8,9,10,3])
se = np.where( arr%2 == 0)
print(se)
```

```
(array([ 1,  3,  5,  6,  8, 10], dtype=int64),)
```

In the above code the index of the elements which are divisibles of 2 and reminder is 0.

In [36]:
```
arr = np.array([1,2,3,4,5,6,4,7,8,9,10,3])
se = np.where( arr%3 == 1)
print(se)
```

```
(array([ 0,  3,  6,  7, 10], dtype=int64),)
```

In the above code the index of the elements which are divisibles of 3 and reminder is 1.

## Search Sorted

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 [47]:
```
arr = np.array([1,2,3,4])
se = np.searchsorted( arr , 3)
print(se)
```

```
2
```

## Search form the Right side

By default the left most index is returned, but we can give side='right' to return the right most index instead.

In [48]:
```
arr = np.array([1,2,3,4])
se = np.searchsorted( arr , 3, side = 'right')
print(se)
```

```
3
```

In first the index is 2 and in second the index is 3 for the element 3 because first search return in left and second is from right.

## Multiple values

To search for the multilpe values more then one ,use an array with the specified values.

In [2]:
```
arr = np.array([1,2,3,4,5,6,4,7,8,9,10,3])
lst = [2,8,6,10]
se = np.searchsorted( arr , lst)
print(se)
```

```
[ 1  8  7 12]
```

The return value is an array : [1 8 7 12] containing the four indexes where 2,8,6,10 would be inserted in the orginal array to maintain the order.

## Numpy sorting arrays

Sorting means putting elements in an ordered sequence. Ordered sequence is any sequence that has an ordered corresponding to elements like numeric or alphabetical , ascending or descending. The Numnpy ndarray object has function called sort() , that will sort a specified array.

In [10]:
```
arr = np.array([1,6,3,2,0,5])
print(np.sort(arr))
```

```
[0 1 2 3 5 6]
```

Note: This method returns a copy of the array, leaving the original array unchanged.

Strings also can be sorted on alphabetical order.

In [12]:
```
arr = np.array(['python',"Numpy",'Pandas',"Data science","Angular"])
print(np.sort(arr))
```

```
['Angular' 'Data science' 'Numpy' 'Pandas' 'python']
```

Boolean Sorting.

In [18]:
```
arr = np.array([True , False , False ,True])
print(np.sort(arr))
```

```
[False False  True  True]
```

## Sorting 2-D arrays

sort() method will sort both arrays on a 2-D array.

In [15]:
```
arr = np.array([[6,4,1,8],[3,5,2,7]])
print(np.sort(arr))
```

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

## Numpy Filter array

Getting some elements out of an existing array and creating a new array out of them is called filtering. In numpy filter an array using a boolean index list. A boolean index list is a boolean corresponding to index in the array. If the value at an index is True that element is included into filtered array and the having index as False is excluded from the filtered array.

In [21]:
```
arr = np.array([2,34,53,23,5,4])
fil = [False , True , True , False , True , False]
newarr = arr[fil]
print(newarr)
```

```
[34 53  5]
```

Look here only the elements of True index is included into newarr , the elements with False index is excluded.

In earlier , see that where and searchsorted for getting index of a element but here we are getting values by giving bool as list of array.

Creating the Filter array. In above code we hard coded the True and False values , but the common use is to create a filter array based on condition.

In [24]:
```
arr = np.array([2,34,53,23,5,4])
fil = []
for e in arr:
    if e > 25:
        fil.append(True)
    else:
        fil.append(False)
newarr = arr[fil]
print(fil)
print(newarr)
```

```
[False, True, True, False, False, False]
[34 53]
```

In [25]:
```
arr = np.array([2,34,53,23,5,4])
fil = []
for e in arr:
    if e % 2 == 0:
        fil.append(False)
    else:
        fil.append(True)
newarr = arr[fil]
print(fil)
print(newarr)
```

```
[False, False, True, True, True, False]
[53 23  5]
```

Creating Filter directly form Array. we can directly substitute the array insted of the iterable varible in our condition and it will just as we expect it to.

In [26]:
```
arr = np.array([2,34,53,23,5,4])
fil = arr > 25
newarr = arr[fil]
print(fil)
print(newarr)
```

```
[False  True  True False False False]
[34 53]
```

In [27]:
```
arr = np.array([2,34,53,23,5,4])
fil = arr % 2 == 0
newarr = arr[fil]
print(fil)
```

```
print(newarr)
```

```
[ True  True False False False  True]
[ 2 34  4]
```

## Random Numbers in Numpy

Random means does not mean different number every time. Random means something that can not predicted logically. Psuedo Random and True Random. Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well. If there is a program to generate random number it can be predicted, thus it is not truly random. Random numbers generated through a generation algorithm are called pseudo random.

Random number Numpy offers the random module with random number with attribute randit()

In [3]:
```
from numpy import random as rd
x = rd.randint(9)
print(x)
```

```
6
```

Random Float The random module's rand() method returns a radnom float 0 and 1.

In [66]:
```
x = rd.rand()
print(x)
```

```
0.43885745646587104
```

Random Array In Numpy we work with arrays , and you can use the two methods from the above example to make random arrays. Inergers The randit() method takes a size parameter where you can specify the shape of an array.

In [70]:
```
x = rd.randint(100 , size =(2))
print(x)
```

```
[55 45]
```

Here we given size as 2 so the random elements also 2.

In [72]:
```
x = rd.randint(100, size =(3,5))
print(x)
print(x.ndim)
```

```
[[ 9 10 88 16 83]
 [14 81 23 42 57]
 [ 1 12 57 10 83]]
2
```

In above output we are getting 2 dimension , 3 arrays and 5 elements each.

## Floats

In [73]:
```
x = rd.rand(5)
print(x)
```

```
[0.91580893 0.61990975 0.09770231 0.70879616 0.18020373]
```

In above output five random elements.

In [74]:
```
x = rd.rand(3,5)
print(x)
```

```
[[0.7475402  0.83953894 0.59985008 0.05853731 0.93203159]
 [0.13323989 0.73171174 0.39748582 0.06157676 0.27923551]
 [0.43479796 0.37957815 0.93735175 0.92718035 0.53190933]]
```

In above output five random elements in 3 different arrays

Random number from a array. The choice() method allows you to generate a random value based on an arrays of values. The choice() method takes an array as a parameter and randomly returns one of the values.

In [77]:
```
x = rd.choice([2,3,4,63,6,3,1,5,6])
print(x)
```

```
63
```

On running code multiple time we get random elements each time.

The choice() method also allows you to return an array of values. WE can add size parameter to specify the shape of the array.

In [10]:
```
x = rd.choice([2,5,7,4], size = (2,3))
```

```
print(x)
```

```
[[7 5 4]
 [2 5 5]]
```

Here two arrays with 3 elements as instructed.

Data Distribution Data distribution is a list of all possible values and , how often each value occurs. Such lists are important when

Mathematical and Statical Functions Numpy provides a comprehensive suite of mathematical function: Basic Arithemetic Operations: Basic addition , multiplication , subration , division ,etc... Statical Function : Functions like np.mean() , np.median() , np.std() , np.var() and np.sum() help summerize the data . Aggregation Function : Function like np.min() , np.max() and np.argamx() return minimum , maximum and the index of the maximum values respectively. Universal Function : unfuncs are the functions that operate element-wise on arrays. They are optimized for performance and can handle arrays of different shapes and sizes.Mathematical Function : np.sin() , np.cos() , np.tan() , np.exp() , np.exp() , np.lig() , np.sqrt() , etc... Comparison Function : np.greater() , np.less() , np.equal() , etc...

In [3]:
```
#Basic Arithematic operation
arr = np.array([1,2,3,4])
arr1 = np.array([5])
print(np.add(arr,arr1))
print(np.subtract(arr,arr1))
print(np.multiply(arr,arr1))
print(np.divide(arr,arr1))
```

```
[6 7 8 9]
[-4 -3 -2 -1]
[ 5 10 15 20]
[0.2 0.4 0.6 0.8]
```

In [4]:
```
#Statistical Function
#In numpy mode function is not avilable directly we have to import scipy library
data = np.array([1,2,3,4])
print("Mean " , np.mean(data))
print("Median " , np.median(data))
print("Standard deviation " , np.std(data))
print("Varince " , np.var(data))
print("Sum " , np.sum(data))
```

```
Mean  2.5
Median  2.5
Standard deviation  1.118033988749895
Varince  1.25
Sum  10
```

In [5]:
```
#Aggregation Function
data = data = np.array([0,1,2,3,4])
print("Maximum value " ,np.max(data))
print("Minimun value " ,np.min(data))
print("index of maximum value " , np.argmax(data))
```

```
Maximum value  4
Minimun value  0
index of maximum value  4
```

In [6]:
```
#Universal Function
#Mathematical function
data = np.array([1,2,3,4])
print("Sign value " , np.sin(data))
print("Cos value " , np.cos(data))
print("Tan value " , np.tan(data))
print("Exponent value " , np.exp(data))
print("Log value " , np.log(data))
print("Sqroot of data " , np.sqrt(data))
```

```
Sign value  [ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
Cos value  [ 0.54030231 -0.41614684 -0.9899925  -0.65364362]
Tan value  [ 1.55740772 -2.18503986 -0.14254654  1.15782128]
Exponent value  [ 2.71828183  7.3890561  20.08553692 54.59815003]
Log value  [0.         0.69314718 1.09861229 1.38629436]
Sqroot of data  [1.         1.41421356 1.73205081 2.        ]
```

Linear Algebra with Numpy Numpy provides a comprehensive set of linear algebra function via the numpy.linalg module.

Linear Algebra with Numpy Numpy provides a comprehensive set of linear algebra functions via the numpy.linalg module . Martix Multiplication : Use np.dot() or the @ operator for matrix multiplication.

In [7]:
```
arr = np.array([[1,2],[4,5]])
arr1 = np.array([[2,4],[5,1]])
print(np.dot(arr , arr1))
#we can use arr @ arr1
```

```
[[12  6]
 [33 21]]
```

Determinant and Inverses. Determinant : Calculate the determinant of a matrix using np.linalg.det() Inverse : Find the inverse of a matrix np.linalg.inv()

In [8]:
```
det_arr = np.linalg.det(arr)
inv_arr = np.linalg.inv(arr)
print(det_arr)
print(inv_arr)
```

```
-2.9999999999999996
[[-1.66666667  0.66666667]
 [ 1.33333333 -0.33333333]]
```

Eigenvalues and Eigenvectors Eigenvalues Defination : Scalars that provide information about the behavior of linear transformation represented by a matrix . Interpretation : When a matrix is multiplied by an eigenvector , the result is simply the eignevector scaled by the eigenvector. Example : If A is the matrix and λ is an eigenvalue , the equation is Av= λv. Here , v is the eigenvector . Eigenvectors : Defination : Non-zero vector that only changed by a scalar factor when a linear transformation is applied to them. Interpretation : They indicate the direction in which the transformation acts. Example : In the above equation , v is an eigenvector corresponding to the eigenvalue λ. Characteristic Polynomial : To find eigenvalues , solve the charactistic polynomial given by : det(A-λI) = 0 where I is the identity matrix

In [9]:
```
A = np.array([[2,3],[1,4]])
ev = np.linalg.eig(A)
print(ev)
```

```
EigResult(eigenvalues=array([1., 5.]), eigenvectors=array([[-0.9486833 , -0.70710678],
       [ 0.31622777, -0.70710678]]))
```

Intution Behind Eigenvalues and Eigenvectors Visual Representation : Imagine applying a transformation represented by a matrix A to a vector.Most vector will change director and length.However , eigenvector are special ; they only get stretched or compressed by the eigenvalue factor without changing their direction. Application : Eigenvalues and eigenvector have significant applications in various fields , including: Principal Component Analysis (PCA) in machine learning for dimensionality reduction. Stability analysis in system of differential equations

In [10]:
```
eigenvalues , eigenvectors = np.linalg.eig(A)
print("eigenvalue value is : " , eigenvalues)
print("eigenvector is  : " , eigenvectors)
```

```
eigenvalue value is :  [1. 5.]
eigenvector is  :  [[-0.9486833  -0.70710678]
 [ 0.31622777 -0.70710678]]
```

Solving Linear System Use np.linalg.solve() to solve a system of linear equation represented Ax = bAx = bAx = b.

In [11]:
```
A = np.array([[2,4],[1,3]])
b = np.array([[5,7],[6,8]])
x = np.linalg.solve(A , b)
print("Linear System : " , x)
```

```
Linear System :  [[-4.5 -5.5]
 [ 3.5  4.5]]
```

Random Number Generation Numpy's random module provides various funstions for generating random numbers, making it essential for simualtions and testing .

Generating Random Numbers: Uniform Distribution : np.random.rand() generates numbers uniformly distributed between (0,1)

In [12]:
```
ud = np.random.rand(3,2)
print(ud)
```

```
[[0.64596292 0.49809286]
 [0.15649399 0.34203981]
 [0.70470069 0.30668736]]
```

Normal Distribution : np.random.randn() generates number form a standard noraml distribution (mean = 0 ,std = 1)

In [13]:
```
nd = np.random.randn(3,2)
print(nd)
```

```
[[ 0.51939973  0.15539193]
 [ 0.29731963  1.02031567]
 [-1.51027413  0.58300421]]
```

Random Integers Generates random integer with np.random.randint(starting , ending , size)

In [14]:
```
rt = np.random.randint(1,10,size = (4,2))
print(rt)
```

```
[[9 9]
 [9 5]
 [3 7]
 [1 4]]
```

Random Sampling Random shuffle an array using np.random.shuffle()

In [15]:
```
arr = np.array([1,2,3,4,5])
np.random.shuffle(arr)
print(arr)
```

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

Splitting arrays using np.split() divides an array into multiple sub-arrays

In [16]:
```
arr = np.array([1,2,3,4,5,6,7,8])
sp = np.split(arr , 4)
print(sp)
```

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