NUMPY (Numerical Pyhton)

In [ ]:
```
1. NumPy (Numerical Pyhton) is a Python library, used for working with arrays.
2. NumPy was created in 2005 by Travis Oliphant.
3. NumPy is used for working with arrays.
4. NumPy aims to provide an array object called ndarray.
5. Numpy provides a lot of supporting functions that make working with ndarray very easy.
6. NumPy is a Python library and is written partially in Python, but most of the parts that require
   fast computation are written in C or C++.
7. NumPy arrays are stored at one continuous place in memory unlike lists,
   so processes can access and manipulate them very efficiently.
   This behavior is called locality of reference in computer science.
   This is the main reason why NumPy is faster than lists.
```

Installation of NumPy

Use the below command to install Numpy:

In [6]:
```
pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\rajee\anaconda3\lib\site-packages (1.23.5)
Note: you may need to restart the kernel to use updated packages.
```

Import NumPy as np

In [2]:
```
# Importing Numpy Library
# 'np' is the generally used alias name for Numpy
# In Python alias are an alternate name for referring to the same thing.

import numpy as np
```

CREATION OF ARRAYS

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 [10]:
```
# creating an ndarray from the list
l = [1,2,3,4,5]
a = np.array(l)
a
```

Out[10]:
```
array([1, 2, 3, 4, 5])
```

In [3]:
```
# creating an ndarray from the Tuple
t = (1,2,3,4,5)
a = np.array(t)
a
```

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

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

0-D arrays

0-D arrays also called Scalars.Scalars are the elements in an array. Each value in an array is a 0-D array.

In [11]:
```
#0D array
arr = np.array(54)
print(arr)
```

```
54
```

1-D Arrays

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

In [14]:
```
#1D array
arr1 = np.array([1,5,2,4,7,8])
print(arr1)
```

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

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.

In [15]:
```
#2D array
arr2 =np.array([[1,2],[3,4],[5,6]])
arr2
```

Out[15]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

In [16]:
```
#3D array
arr3 = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
arr3
```

Out[16]:
```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

type() : Method to determine the type of a given variable.

In [4]:
```
# type() method to determine the type of a given variale.

a = np.array([10,20,30,40,50]) # variale 'a' is ndarray object.
type(a)
```

Out[4]:
```
numpy.ndarray
```

Data Types in NumPy

In [ ]:
```
Below is a list of all data types in NumPy and the characters used to represent them.

i - integer
b - boolean
u - unsigned integer
f - float
```

```
c - complex float
m - timedelta
M - datetime
O - object
S - string
U - unicode string
V - fixed chunk of memory for other type ( void )
```

Creating Arrays With a Defined Data Type

In [9]:
```
# To define the data type of array elements
# To create an array  with data type Float

c =np.array([1,2,3,4],dtype='f')
# c = np.array([1,2,3,4],dtype='U1') #To represent datas as string with 1 character
c
```

Out[9]:
```
array([1., 2., 3., 4.], dtype=float32)
```

In [8]:
```
# To deine type of array elements as String
# In U1 : U -String data type , 1- No. of charactres in String (if U2: means string with 2 charactres)

arr = np.array(['agg','bdd','cdd','dsd'])
arr
```

Out[8]:
```
array(['agg', 'bdd', 'cdd', 'dsd'], dtype='<U3')
```

In [19]:
```
# To deine type os array elements as String
# In U1 : U - String data type , 1- No.of charactres in Strring (if U2 : means string with 2 charactres)

c = np.array(['agg','bdd','cdd','dsd'],dtype='U2')
c
```

Out[19]:
```
array(['ag', 'bd', 'cd', 'ds'], dtype='<U2')
```

What if a Value Can Not Be Converted?

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

In [12]:
```
c = np.array(['agg','bdd','cdd','dsd'],dtype='i')
c # here, array data type is <U3  and it can not be converted to integer type, i. So the program will throw a ValueErro
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[12], line 1
----> 1 c = np.array(['agg','bdd','cdd','dsd'],dtype='i')
      2 c

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

Checking the Data Type of an Array

In [ ]:
```
dtype  : The NumPy array object has a property called 'dtype' that returns the data type of the array.
```

In [11]:
```
#Checking the Data Type of an Array

c = np.array(['agg','bdd','cdd','dsd'])
print(c.dtype) # return U3 (String data type with 3 characters)
```

```
<U3
```

NumPy Array Copy vs View

In [ ]:
```
copy()   : 1. Method to create a copy of an existing (original) array.
           2. The copy owns the data and any changes made to the copy will not affect original array,
              and any changes made to the original array will not affect the copy.

view()   : 1. Mehtod to create a view of a original array.
           2. The view does not own the data and any changes made to the view will affect the original array,
              and any changes made to the original array will affect the view.
```

copy() Mehtod

In [14]:
```
#Make a copy, change the original array, and display both arrays:

arr = np.array(['a','b','c','d'])
c   = arr.copy()
c[0] = 'A' # changed the 0th index value as 42 in the copy

print(arr) #original array
print(c)   # copy
```

```
['a' 'b' 'c' 'd']
['A' 'b' 'c' 'd']
```

view() Mehtod

In [15]:
```
#Make a copy, change the copy array, and display both arrays:

arr = np.array(['a','b','c','d'])
c   = arr.view()
c[0] = 'A' # changed the 0th index value as 4 in the view

print(arr) #original array
print(c)   # copy
```

```
['A' 'b' 'c' 'd']
['A' 'b' 'c' 'd']
```

Creating arrays with different Dimensions (Excercise)

In [21]:
```
#Creating 0-D Array
d0 = np.array([12])
d0
```

Out[21]:
```
array([12])
```

In [22]:
```
# Creating 1-D array
d1 = np.array([1,2,3,4])
d1
```

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

In [27]:
```
# Creating 2-D array (array elements should be 1d array, contains 2 square brackets)
d2 = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
d2
```

Out[27]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [28]:
```
# Creating 3-D array -  (array elements should be 2d array, contains three square brackets)
d3 = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
d3
#type(d3)
```

Out[28]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

In [32]:
```
# Creating arrays with different dimensions.

a = np.array(42) # 0D array
b = np.array([1, 2, 3, 4, 5]) #1D array
c = np.array([[1, 2, 3], [4, 5, 6]]) # 2D array
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]]) #3D array
print(a)
print(b)
print(c)
print(d)
```

```
42
[1 2 3 4 5]
[[1 2 3]
 [4 5 6]]
[[[1 2 3]
  [4 5 6]]

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

Checking the Data Type of an Array (Excercise)

In [16]:
```
a = np.array(42)
a.dtype
```

Out[16]:
```
dtype('int32')
```

In [34]:
```
# checking data types
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)  # U6 U- String 6 - Maximum number of character in the given list of strings
```

```
<U6
```

In [52]:
```
a = np.array([1,2.0,3,4,])
a.dtype
#a
```

Out[52]:
```
dtype('float64')
```

In [39]:
```
a = np.array(['apple','banana','orange','Watermelon'])
a.dtype
```

Out[39]:
```
dtype('<U10')
```

In [54]:
```
a = np.array(['apple','banana',1,89,5.0,'Watermelon'])
a.dtype
#a
```

Out[54]:
```
dtype('<U32')
```

Dimensions of the array

ndim - attribute used to identify the dimension of the given array.

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

Out[3]:
```
1
```

In [6]:
```
c = np.array([[1,2,3],[1,2,3]])
c.ndim
```

Out[6]:
```
2
```

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

Out[8]:
```
3
```

Define the number of dimensions

ndmin - attribute used to define the dimension of array.

In [58]:
```
# Creating higher dimensional arrays

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

```
print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[1 2 3 4]]]]
number of dimensions : 4
```

In [19]:
```
# Creating higher dimensional arrays example
a = np.array([1,2,3,4],ndmin=32) #maximum valuue for ndmin is 32,We can make up to 32 dimension array
a
```

Out[19]:
```
array([[[[[[[[[[[[[[[[[[[[[[[[[[[[[[1, 2,
                                     3, 4]]]]]]]]]]]]]]]]]]]]]]]]]]]]])
```

CHECKING SHAPE

In [ ]:
```
Shape of an Array : The shape of an array is the number of elements in each dimension.

shape  : 1. Attribute used to determine the shape of an array.
         2. Returns a tuple with each index having the number of corresponding elements.
```

In [20]:
```
# checking the shape
c = np.array([[1, 2, 3], [4, 5, 6]])
c.shape
```

Out[20]:
```
(2, 3)
```

In [21]:
```
a = np.array([1,2,3,4])
a.shape
```

Out[21]:
```
(4,)
```

In [23]:
```
a = np.array([[1,2,3],[4,5,6]])
a.shape
```

Out[23]:
```
(2, 3)
```

In [25]:
```
a = np.array([[[1,2],[3,4]],[[5,6],[7,8]],[[9,10],[11,12]]])
a.shape
```

Out[25]:
```
(3, 2, 2)
```

Size of an array

size : Attribute used to determine the size of an array. Size returns the number of elements in the array.

In [28]:
```
# size - returns the no. of elements in the array

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

Out[28]:
```
12
```

In [29]:
```
a = np.array([[[1,2],[3,4]],[[5,6],[7,8]],[[9,10],[11,12]]])
a.size
```

Out[29]:
```
12
```

In [30]:
```
a = np.array([[1,2,3],[4,5,6]])
a.size
```

Out[30]:
```
6
```

EXERCISE

In [36]:
```
# check the number of dimensions

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

```
1
```

In [35]:
```
# create 2D array from 1D array

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

Out[35]:
```
array([[1, 2, 3, 4]])
```

Array creation with homogenious data.

In [ ]:
```
Methods used to create arrays having homogenious data are:
    1. zeros()   : Method to create array of zeros. This method takes array shape as arguments.
    2. ones()    : Method to create array of ones. This method takes array shape as arguments.
    3. full()    : Method used to create an array with a single value.
```

In [37]:
```
#Creating array of zeros

a=np.zeros((2,3),dtype='int')
a
```

Out[37]:
```
array([[0, 0, 0],
       [0, 0, 0]])
```

In [40]:
```
a = np.zeros((2,1),dtype='float')
a
```

Out[40]:
```
array([[0.],
       [0.]])
```

In [4]:
```
#creating array of 1's

b= np.ones((3,2), dtype = 'int')
b
```

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

In [5]:
```
a = np.ones((2,3,4))
a
```

Out[5]:
```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

In [42]:
```
#creating array with a single value

c=np.full((2,3),'abc')
c
```

Out[42]:
```
array([['abc', 'abc', 'abc'],
       ['abc', 'abc', 'abc']], dtype='<U3')
```

Random Numbers in NumPy

In [ ]:
```
NumPy offers the random module to work with random numbers.
The random module  methods:
    1. rand()      : Method returns a random float between 0 and 1.
    2. randint(n)  : Method to generate a random integer from 0 to n.
```

In [9]:
```
#Generate a random integer from 0 to 50:(randint() method)

from numpy import random # command to import random module
x = random.randint(50)
print(x)
```

```
33
```

In [7]:
```
#Generate a random float from 0 to 1: (rand() method)

from numpy import random # command to import random module
r = random.rand()
print(r)
```

```
0.9211259218361701
```

Generate Random Array

In [ ]:
```
There are several methods to generate random arrays:

    1. randint()  : Create an array of integer numbers. Method takes a size parameter where you can specify
                    the shape of an array.

    2. rand()     : Create an array of float numbers. Method also allows you to specify the shape of the array.

    3. random()   : Create an array of random numbers.

    4. choice()   : Method allows us to generate a random value based on an array of values.
                    The choice() method takes an array as a parameter and randomly returns one of the values.
                    It also allows us to return an array of values, add a size parameter to specify the shape of the ar
```

In [14]:
```
#Generate a 1-D array using randint() method containing 6 random integers from 0 to 50:

arr = np.random.randint(50, size=(6))
print(arr)
```

```
[25 27 36 30 34 29]
```

In [15]:
```
#Generate a 2-D array using randint() method containing 8 random integers from 0 to 50:

arr = np.random.randint(50, size=(2,4))
print(arr)
```

```
[[42 39 36 23]
 [32 45  9  8]]
```

In [20]:
```
#Generate a 1-D array containing 5 random floats:

arr = random.rand(5)
print(arr)
```

```
[0.88120064 0.26516753 0.58133605 0.35433232 0.52485111]
```

In [22]:
```
#Generate a 2-D array containing 4 random floats:

arr = random.rand(2,2)
print(arr)
```

```
[[0.52490195 0.82658445]
 [0.11729888 0.09937123]]
```

In [25]:
```
#creating array with random values

arr = np.random.random((5,5))
print(arr)
```

```
[[0.52944491 0.26274302 0.06843229 0.09822787 0.02160015]
 [0.88490897 0.12273519 0.38431442 0.7932834  0.64065968]
 [0.9012493  0.59756857 0.16803969 0.98383741 0.78738222]
 [0.16978969 0.96619817 0.73611989 0.32924178 0.38877664]
 [0.69260251 0.56476599 0.50728818 0.83963856 0.32862201]]
```

In [27]:
```
#generate an array using choice() method

x = random.choice([6,2,8,1])
print(x)
```

```
8
```

In [31]:
```
#Generate a 2-D array that consists of the values in the array parameter (10,20,30,40):

from numpy import random
```

```
arr = random.choice([10,20,30,40], size=(3,2))
print(arr)
```

```
[[40 20]
 [40 40]
 [10 10]]
```

ARRAY CREATION - NUMERICAL RANGES

In [46]:
```
#creating array with step size arange(start,end,step)
e=np.arange(1,10,2)
e
```

Out[46]:
```
array([1, 3, 5, 7, 9])
```

In [49]:
```
#creating array with equal spacing
f=np.linspace(1,10,2)
f
```

Out[49]:
```
array([ 1., 10.])
```

Numpy array Indexing

Two types:

1. Positive indexing 2. Negative indexing

Access Array Elements:

1. Positive indexing You can access an array element by referring to its index number. In positive indexing, The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.
2. Negative Indexing Use negative indexing to access an array from the end.

POSITIVE INDEXING

Accessing 1D Array

In [22]:
```
# accessing 1-D
# access first element
arr = np.array([1, 2, 3, 4,5])
print(arr[0])
```

```
1
```

In [24]:
```
# to access second element
print(arr[1])
```

```
2
```

In [25]:
```
# get 3rd and 5 th and add them
print(arr[2] + arr[4])
```

```
8
```

Accessing 2D Array

2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

In [27]:
```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
arr
```

Out[27]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

In [51]:
```
# to access element on the first row and second column
print('2nd element on 1st row: ', arr[0, 1])
```

```
2nd element on 1st row:  2
```

In [28]:
```
#access element on the 2nd row and 5th column
print('5th element on 2nd row: ', arr[1, 4])
```

```
5th element on 2nd row:  10
```

Accessing 3D Array

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

In [29]:
```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr
```

Out[29]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

In [30]:
```
# accessing 3-d (dimensions and index )
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
# access 3rd of second array of the first array
print(arr[0, 1, 2])
```

```
6
```

In [57]:
```
# accessing whole row
arr[0,0]
```

Out[57]:
```
array([1, 2, 3])
```

Negative Indexing

In [59]:
```
arr[-1, -2, -1]
```

Out[59]:
```
9
```

EXERCISE

In [4]:
```
#print first item in the array

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

```
1
```

In [5]:
```
# print 50

arr = np.array([[10, 20, 30, 40], [50, 60, 70, 80]])
arr[1,0]
```

Out[5]:
```
50
```

In [7]:
```
# print 11

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

Out[7]:
```
11
```

NumPy Array Slicing

In [ ]:
```
# Array slicing
# Slicing 1-d array [start:end] [start:end:step]
```

In [9]:
```
# slicing from index 1 to 5
arr = np.array([1,2,4,6,3,7,4])
arr[0:6]
```

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

In [10]:
```
#slices from index 4 to the end
arr[4:]
```

Out[10]:
```
array([3, 7, 4])
```

In [11]:
```
# beginning to index 4
arr[0:5]
```

Out[11]:
```
array([1, 2, 4, 6, 3])
```

SLICING WITH NEGATIVE INDEX

In [12]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
# slicing from index -3 to index -1
arr[-3:]
```

Out[12]:
```
array([5, 6, 7])
```

In [15]:
```
# every other element from index 1 to index 5
print(arr[1:6:2])
```

```
[2 4 6]
```

In [16]:
```
# every other element from entire array
arr[::2]
```

Out[16]:
```
array([1, 3, 5, 7])
```

SLICING 2D ARRAY

In [19]:
```
# slicing 2-D array
# From the second element, slice from index 1 to index 4
```

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

```
[ 7  8  9 10]
```

In [23]:
```
# from both array return index 2
print(arr2[0:2,2])
#print(arr2[0:, 2])
```

```
[3 8]
```

In [27]:
```
# from both elements slice 1 - 4(returns a 2 d array)

arr2[0:,1:5]
```

Out[27]:
```
array([[ 2,  3,  4,  5],
       [ 7,  8,  9, 10]])
```

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

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

In [28]:
```
# From the second element, slice from index 1 to index 4
arr2[1,1:5]
#print(arr[1, 1:5])
```

Out[28]:
```
array([ 7,  8,  9, 10])
```

In [29]:
```
# from both return index 2
arr2[0:,2]
```

Out[29]:
```
array([3, 8])
```

EXERCISE

In [30]:
```
#1.Everything from (including) the second item to
#(not including) the fifth item.

arr = np.array([10, 15, 20, 25, 30, 35, 40])
print(arr[1:4])
```

```
[15 20 25]
```

In [31]:
```
#3.Every other item from (including) the second item to
#(not including) the fifth item.

arr[1::2]
```

Out[31]:
```
array([15, 25, 35])
```

In [32]:
```
#4.Every other item from the entire array.
arr[::2]
```

Out[32]:
```
array([10, 20, 30, 40])
```

In [ ]:
```
arr = np.array([10, 15, 20, 25, 30, 35, 40])
print(arr   )
```

CHECKING THE SHAPE:

In [19]:
```
# checking the shape of array
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

```
(2, 4)
```

In [33]:
```
arr = np.array([1,2,3,4,5,6])
arr.shape
```

Out[33]:
```
(6,)
```

In [34]:
```
arr = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
arr.shape
```

Out[34]:
```
(2, 2, 3)
```

Array Reshape

In [ ]:
```
1. Reshaping means changing the shape of an array.

2. The shape of an array is the number of elements in each dimension.

3. By reshaping we can add or remove dimensions or change number of elements in each dimension.
```

In [20]:
```
# reshaping an 1D array to 2D array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)

print(newarr)
```

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

In [17]:
```
#Reshaping 1-D array with 12 elements into a 3-D array.

#The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:


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

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

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

In [24]:
```
# Can we reshape higher dimension to lower dimension"

arr = np.array([[1,2,3,4],[6,7,8,9]])
arr.reshape(8) #2d to 1d array
```

Out[24]:
```
array([1, 2, 3, 4, 6, 7, 8, 9])
```

In [19]:
```
arr = np.array([[[1,2,3],[4,5]],[[1,2,8],[3,4,9,10]]],dtype=object) # when you want to give diffent no. of items
                                                                    # in each array declare dtype= object
arr
```

Out[19]:
```
array([[list([1, 2, 3]), list([4, 5])],
       [list([1, 2, 8]), list([3, 4, 9, 10])]], dtype=object)
```

In [22]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8,9,10,11,12])
newarr = arr.reshape(4, 3)
print(newarr)
```

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

Flattening the arrays

In [ ]:
```
1. Flattening array means converting a multidimensional array into a 1D array.

reshape(-1)   : Method to use converting a multidimensional array into a 1D array.
```

In [18]:
```
#Convert the 2D array into a 1D array:

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

newarr = arr.reshape(-1)

print(newarr)
```

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

Array Iterating

Iterating means going through elements one by one. We can use for loop to iterate through array elements.

In [24]:
```
# iterating(1D)
arr = np.array([1, 2, 3])

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

```
1
2
3
```

In [28]:
```
arr = np.array([3,6,8,9,4])
for i in arr:
    print(i)
```

```
3
6
8
9
4
```

In [25]:
```
# iterating 2-D
arr = np.array([[1, 2, 3], [4, 5, 6]])

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

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

In [26]:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    for y in x:
        print(y)
```

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

In [29]:
```
arr = np.array([[1,2,3],[4,5,6]])
for i in arr:
    print(i)
```

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

In [31]:
```
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
```

In [27]:
```
#iterating 3-d arrays
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
    print(x)
```

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

In [28]:
```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
    for y in x:
        for z in y:
            print(z)
```

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

In [35]:
```
arr = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
for i in arr:
    print("\n",i)
```

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

 [[5 6]
 [7 8]]
```

In [37]:
```
arr = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
for i in arr:
    for j in i:
        print("\n",j)
```

```
[1 2]

[3 4]

[5 6]

[7 8]
```

In [39]:
```
arr = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
for i in arr:
    for j in i:
        for k in j:
            print("\n",k)
```

```
1

2

3

4

5

6

7

8
```

nditer() - Effectively iterate over multidimensional array elements.

In [29]:
```
# iterating using nditer
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

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

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

In [40]:
```
arr = np.array([[4,7],[3,8],[1,9]])
for i in np.nditer(arr):
    print(i)
```

```
4
7
3
8
1
9
```

Joining NumPy Arrays

In [ ]:
```
1. Joining means putting contents of two or more arrays in a single array.
2. In SQL we join tables based on a key, whereas in NumPy we join arrays by axes
3. Methods used in numpy to join arrays are:

    concatenate()  : We pass a sequence of arrays that we want to join to the concatenate() function, along with the ax
                     If axis is not explicitly passed, it is taken as 0.

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

    vstack()       : Method to stack along columns.

    hstack()       : Method to stack along rows.

    dstack()       : Method to stacking Along Height (depth).
```

join arrays using concatenate()

In [35]:
```
# joining 1D arrays

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

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

print(arr)
```

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

In [79]:
```
# Joining 2D arrays

arr1 = np.array([[1,2],[3,4]])
arr2 = np.array([[5,6],[7,8]])
#arr = np.concatenate((arr1,arr2))
#arr = np.hstack((arr1,arr2))
#arr =np.vstack((arr1,arr2))
arr =np.stack((arr1,arr2))
#arr = np.dstack((arr1, arr2))
print(arr)
print("Shape:", arr.shape)
print("Shape:", arr1.shape)
```

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

 [[5 6]
  [7 8]]]
Shape: (2, 2, 2)
Shape: (2, 2)
```

In [45]:
```
# joining rows 2d

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

arr2 = np.array([[5, 6], [7, 8]])

#arr  = np.concatenate((arr1, arr2),axis=0)
arr  = np.concatenate((arr1, arr2),axis=1)
print(arr)
```

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

In [43]:
```
# joining rows 3d
arr1 = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])

arr2 = np.array([[[9,10],[13,14]],[[15,16],[17,18]]])

#arr  = np.concatenate((arr1, arr2),axis=0)
#arr  = np.concatenate((arr1, arr2),axis=1)
arr  = np.concatenate((arr1, arr2),axis=2)

print(arr)
```

```
[[[ 1  2  9 10]
  [ 3  4 13 14]]

 [[ 5  6 15 16]
  [ 7  8 17 18]]]
```

Join arrays using Stack()

In [51]:
```
#stacking

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=0)
#arr = np.stack((arr1, arr2), axis=1)

print(arr)
```

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

Join arrays using hstack()

In [48]:
```
#Horizontal Stacking, stacks one beside other

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

arr = np.hstack((arr1, arr2))
print(arr)
```

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

Join arrays using vstack()

In [72]:
```
#Vertical Stacking stacks one above other

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

arr = np.vstack((arr1, arr2))
#arr = np.concatenate((arr1,arr2))
#arr = np.hstack((arr1,arr2))
arr
```

Out[72]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Join arrays using dstack()

In [35]:
```
#stacking along depth

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

arr = np.dstack((arr1, arr2))
arr
```

Out[35]:
```
array([[[1, 4],
        [2, 5],
        [3, 6]]])
```

Exercise

In [58]:
```
# Use a correct NumPy method to join two arrays into a single array.

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))
arr
```

Out[58]:
```
array([1, 2, 3, 4, 5, 6])
```

Splitting NumPy Arrays

In [ ]:
```
1. Splitting is reverse operation of Joining.
2. Joining merges multiple arrays into one and Splitting breaks one array into multiple.
3. Methods used in numpy to join arrays are:

    array_split()   : We can use array_split() for splitting arrays, we pass it the array we want to split and
                      the number of splits. The return value of the array_split() method is an array containing
                      each of the split as an array.

    hsplit()        : Method used to split the array along horizontal axis (rows).

    vsplit()        : Method used to split the array along vertical axis (columns).
```

Split array using arraysplit()

In [59]:
```
# split arrays using arraysplit() method

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

# splits in 3 parts
newarr = np.array_split(arr, 3)

print(newarr)
```

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

In [52]:
```
arr = np.array([1, 2, 3, 4, 5, 6])

# splits in 4 parts

newarr = np.array_split(arr,5)
print(newarr)
```

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

In [57]:
```
# Splitting 2-dimensional array into two 2D array.

a = np.array([[1,2,3,4,5,6],[7,8,9,10,11,12]])
newarr = np.array_split(a,2)
```

```
print(newarr)
```

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

Split array using hsplit()

In [39]:
```
# horizontal splitting

a = np.array([[1,2,3,4,5,6],[7,8,9,10,11,12]])
print("Splitting along horizontal axis into 2 parts:\n", np.hsplit(a, 2))
```

```
Splitting along horizontal axis into 2 parts:
 [array([[1, 2, 3],
       [7, 8, 9]]), array([[ 4,  5,  6],
       [10, 11, 12]])]
```

Split array using vsplit()

In [40]:
```
#Vertical Split

a = np.array([[1,2,3,4,5,6],[7,8,9,10,11,12]])
print("Splitting along Vertical axis into 2 parts:\n", np.vsplit(a, 2))
```

```
Splitting along Vertical axis into 2 parts:
 [array([[1, 2, 3, 4, 5, 6]]), array([[ 7,  8,  9, 10, 11, 12]])]
```

Searching Arrays

In [ ]:
```
where()  : Method to search  elements in the array.
           Return array of index(s) of searched item.
```

In [41]:
```
# search using where() method

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

x = np.where(arr == 4)

print(x) # Return the array of index position(s) of searched item
```

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

In [61]:
```
#Searching items that has more than one occurences.

arr = np.array([1,4,8,3,1,5,8,4])
x =np.where(arr==8)
print (x)
```

```
(array([2, 6], dtype=int64),)
```

In [64]:
```
#Searching item 7

arr = np.array([1,4,8,3,1,7,8,4,7,7,7])
x =np.where(arr==7)
print (x)
```

```
(array([ 5,  8,  9, 10], dtype=int64),)
```

In [42]:
```
# search for even number

x = np.where(arr%2 == 0)
x
```

Out[42]:
```
(array([1, 3, 5, 6], dtype=int64),)
```

In [43]:
```
# search for odd number

x = np.where(arr%2 == 1)
x
```

Out[43]:
```
(array([0, 2, 4], dtype=int64),)
```

Sorting Arrays

In [ ]:
```
sort() :  Method to sort the array elements.
```

In [66]:
```
# sort the array elements

arr = np.array([3, 2, 0, 1,-1, 10,-8])
sarr =np.sort(arr)
print(sarr)
```

```
[-8 -1  0  1  2  3 10]
```

In [68]:
```
# Sorting of 2D array

arr = np.array([[41,50,8],[31,77,50],[23,90,12]])
sarr = np.sort(arr)
sarr
```

Out[68]:
```
array([[ 8, 41, 50],
       [31, 50, 77],
       [12, 23, 90]])
```

In [69]:
```
# sorts alphabetically whrn you get an string of arrays as input.

arr = np.array(['Banana', 'Cherry', 'Apple'])

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

```
['Apple' 'Banana' 'Cherry']
```

End