Introduction

NumPy stands for ‘Numerical Python’. It is a package for data analysis and scientific computing with Python. NumPy uses a multidimensional array object, and has functions and tools for working with these arrays. The powerful n-dimensional array in NumPy speeds-up data processing. History: NumPy was created by Travis Oliphant in 2005, while he was a graduate student at the Mayo Clinic.

Installing NumPy

NumPy can be installed by typing following command: pip install NumPy

Array

We have learnt about various data types like list, tuple, and dictionary. In this chapter we will discuss another datatype ‘Array’. An array is a data type used to store multiple values using a single identifier (variable name). An array contains an ordered collection of data elements where each element is of the same type and can be referenced by its index (position). The important characteristics of an array are:

Each element of the array is of same data type, though the values stored in them may be differet.n

The entire array is stored contiguously in memory. This makes operations on array fas.t

Each element of the array is identified or referred using the name of the Array along wit the index of that element, which is unique f each eleen

t. The index of an element isn integral value associated with the elemt, based on the element’s position in the array.

Difference Between List and Array

List List can have elements of different data types for example, [1,3.4, ‘hello’, ‘a@’] Elements of a list are not stored contiguously in memory. Lists do not support element wise operations, for example, addition, multiplication, etc. because elements may not be of same tye.p Lists can contain objects of different datatype that Python must store the typ information for every element along with i element vlu e. Thus lists take more spe in memory and are less effici List is a part of core Python. Array All elements of an array are of same data type for example, an array of floats may be: [1.2, 5.4, 2.7] Array elements are stored in contiguous memory locations. This makes operations on arrays faster than lists. Arrays support element wise operations. For example, if A1 is an array, it is possible to say A1/3 to divide each element of the array by 3. NumPy array takes up less space in memory as compared to a list because arrays do not require to store datatype of each element separately. Array (ndarray) is a part of NumPy library.ble>

Creation of NumPy Arrays from List

In [2]:
```python
# There are several ways to create arrays. To create an array

import numpy as np
# NumPy is loaded as np (we can assign any name), numpy must
```

In [3]:
```python
# The NumPy’s array() function converts a given list into an

array = np.array([10, 20, 30])
array
```

Out[3]:
```
array([10, 20, 30])
```

Creating a Zero-Dimensional Array

In [5]:
```python
arrZero = np.array(80)
print(arrZero)

print(arrZero.ndim)
```

```
80
0
```

Creating a 1-D Array

An array with only single row of elements is called 1-D array. Let us try to create a 1-D array from a list which contains numbers as well as strings.

In [10]:
```python
array1 = np.array([5, -7.4, 'a', 7.2])
array1

# Observe that since there is a string value in the list, all
# Note: U32 means Unicode-32 data type.
```

Out[10]:
```
array(['5', '-7.4', 'a', '7.2'], dtype='<U32')
```

Creating a 2-D Array

In [15]:
```python
array2 = np.array([[2.4, 3],
                   [4.9, 7],
                   [0, -1]])
array2

# Observe that the integers 3, 7, 0 and -1 have been promoted
```

Out[15]:
```
array([[ 2.4,  3. ],
       [ 4.9,  7. ],
       [ 0. , -1. ]])
```

Attributes of NumPy Array

.ndim: Gives the number of dimensions of the array as an integer value. Arrays can be 1-D, 2-D or n-D.

In [13]:
```python
print(array1.ndim)
print(array2.ndim)
```

```
1
2
```

.shape: It gives the sequence of integers indicating the size of the array for each dimension.

In [18]:
```
print(array.shape)
print(array1.shape)
print(array2.shape)
```

```
(3,)
(4,)
(3, 2)
```

.size: It gives the total number of elements of the array. This is equal to the product of the elements of shape.

In [19]:
```
print(array.size)
print(array1.size)
print(array2.size)
```

```
3
4
6
```

.dtype: is the data type of the elements of the array. All the elements of an array are of same data type. Common data types are int32, int64, float32, float64, U32, etc.

In [20]:
```
print(array.dtype)
print(array1.dtype)
print(array2.dtype)
```

```
int32
<U32
float64
```

.itemsize: It specifies the size in bytes of each element of the array. Data type int32 and float32 means each element of the array occupies 32 bits in memory. 8 bits form a byte. Thus, an array of elements of type int32 has itemsize 32/8=4 bytes. Likewise, int64/float64 means each item has itemsize 64/8=8 bytes.

In [21]:
```
print(array.itemsize)
print(array1.itemsize)
print(array2.itemsize)
```

```
4
128
8
```

Other Ways of Creating NumPy Arrays

1. We can specify data type (integer, float, etc.) while creating array using dtype as an argument to array(). This will convert the data automatically to the mentioned type. In the following example, nested list of integers are passed to the array function. Since data type has been declared as float, the integers are converted to floating point numbers.

In [23]:
```
array3 = np.array([[1, 2], [3, 4]], dtype=float)

array3
```

Out[23]:
```
array([[1., 2.],
       [3., 4.]])
```

2. We can create an array with all elements initialised to 0 using the function zeros(). By default, the data type of the array created by zeros() is float. The following code will create an array with 3 rows and 4 columns with each element set to 0.

In [30]:
```
array4 = np.zeros((3,2))
array4
```

Out[30]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

3. We can create an array with all elements initialised to 1 using the function ones(). By default, the data type of the array created by ones() is float. The following code will create an array with 3 rows and 2 columns.

```
In [32]: array5 = np.ones((3,2), dtype=int)
         array5
```

```
Out[32]: array([[1, 1],
                [1, 1],
                [1, 1]])
```

```
In [6]: np.ones((6, 3, 3))   # Here 6 is no. of matrix to created 3
```

```
Out[6]: 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.],
                [1., 1., 1.]],

               [[1., 1., 1.],
                [1., 1., 1.],
                [1., 1., 1.]],

               [[1., 1., 1.],
                [1., 1., 1.],
                [1., 1., 1.]],

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

4. We can create an array with all elements initialised to any int using the function .full(). By default, the data type of the array created by full() is the data type of initialised value. The following code will create an array with 4 rows and 4 columns.

```
In [12]: np.full((4, 4),45)
```

```
Out[12]: array([[45, 45, 45, 45],
                [45, 45, 45, 45],
                [45, 45, 45, 45],
                [45, 45, 45, 45]])
```

```
In [8]: np.full((40,40),678)
```

```
Out[8]: array([[678, 678, 678, ..., 678, 678, 678],
               [678, 678, 678, ..., 678, 678, 678],
               [678, 678, 678, ..., 678, 678, 678],
               ...,
               [678, 678, 678, ..., 678, 678, 678],
               [678, 678, 678, ..., 678, 678, 678],
               [678, 678, 678, ..., 678, 678, 678]])
```

```
In [13]: # importing random
         from numpy import random

         x = random.randint(100)
         print(x)
```

```
15
```

```
In [10]: np.random.rand(4,4)
```

```
Out[10]: array([[0.18987916, 0.69608046, 0.63399454, 0.41937221],
                [0.60875462, 0.20413715, 0.32916978, 0.00849296],
                [0.95772644, 0.35098137, 0.1940079 , 0.03414478],
                [0.8351916 , 0.75436728, 0.67406473, 0.21766497]])
```

5. We can create an array with numbers in a given range and sequence using the arange() function. This function is analogous to the range() function of Python.

array6 = np.arange(6) array6

```
In [34]: array7 = np.arange(-2, 24, 4)
         array7
```

```
Out[34]: array([-2,  2,  6, 10, 14, 18, 22])
```

Indentity Matrix

```
In [14]: np.identity(5)
```

```
Out[14]: 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.]])
```

Indexing and Slicing

NumPy arrays can be indexed, sliced and iterated over.

Indexing

Marks of Students in different Subjects

Name	Maths	English	Science
Ramesh	78	67	56
Vedika	76	75	47
Harun	84	59	60
Prasad	67	72	54

```
In [35]: marks = np.array([[78, 67, 56],
                           [76, 75, 47],
                           [84, 59, 60],
                           [67, 72, 54]])

         marks
```

```
Out[35]: array([[78, 67, 56],
                [76, 75, 47],
                [84, 59, 60],
                [67, 72, 54]])
```

Example: Columns 0 1 2 3 4 [[1 2 3 4 5] => 0th Row [6 7 8 9 10]] => 1st Row

```
In [37]: # accesses the element in the 1st row in the 3rd column

         marks[0, 2]
```

```
Out[37]: 56
```

```
In [39]: marks[0, 4]   # index Out of Bound "Index Error".
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most rec
ent call last)
Cell In[39], line 1
----> 1 marks[0, 4]   # index Out of Bound "Index Error".

IndexError: index 4 is out of bounds for axis 1 with size 3
```

Slicing

Sometimes we need to extract part of an array. This is ddone through slicing. We can define which part of the array to be sliced by specifying the start and end inde values usi\ng [start : -1end] along with the array name.

```
In [41]: array7
```

```
Out[41]: array([-2,  2,  6, 10, 14, 18, 22])
```

```
In [42]: array7[3:5]
```

```
Out[42]: array([10, 14])
```

```
In [43]: array7[ : : -1]
```

```
Out[43]: array([22, 18, 14, 10,  6,  2, -2])
```

```
In [44]: array8 = np.array([[-7, 0, 10, 20],
                            [-5, 1, 40, 200],
                            [-1, 1, 4, 30]])
         array8
```

```
Out[44]: array([[ -7,   0,  10,  20],
                [ -5,   1,  40, 200],
                [ -1,   1,   4,  30]])
```

```
In [45]: array8[0:3,2]
```

```
Out[45]: array([10, 40,  4])
```

```
In [48]: # Another way

         array8[:,2]
```

```
Out[48]: array([10, 40,  4])
```

```
In [46]: array8[1:3,0:2]
```

```
Out[46]: array([[-5,  1],
                [-1,  1]])
```

Operations on Array

Arithmetic Operations

```
In [64]: array1 = np.array([[3,6],[4,2]])
         array2 = np.array([[10,20],[15,12]])

         print(array1)
         print()
         print(array2)
```

```
[[3 6]
 [4 2]]

[[10 20]
 [15 12]]
```

```
In [65]: # Element-wise addition of two matrices.

         array1 + array2
```

```
Out[65]: array([[13, 26],
                [19, 14]])
```

```
In [60]: # Subtraction

         array1 - array2
```

```
Out[60]: array([[ -7, -14],
                [-11, -10]])
```

```
In [61]: # Multiplication
```

```
array1 * array2
```

```
Out[61]: array([[ 30, 120],
                [ 60,  24]])
```

```
In [66]: # Matrix Multiplication

         array1 @ array2
```

```
Out[66]: array([[120, 132],
                [ 70, 104]])
```

```
In [67]: # Exponentiation

         array1 ** 3
```

```
Out[67]: array([[ 27, 216],
                [ 64,   8]], dtype=int32)
```

```
In [68]: # Element wise Remainder of Division (Modulo)

         array2 % array1
```

```
Out[68]: array([[1, 2],
                [3, 0]])
```

Transpose

```
In [70]: array3 = np.array([[10, -7, 0, 20],
                            [-5, 1, 200, 40],
                            [30, 1, -1, 4]])
         array3
```

```
Out[70]: array([[ 10,  -7,   0,  20],
                [ -5,   1, 200,  40],
                [ 30,   1,  -1,   4]])
```

```
In [71]: # the original array does not change

         array3.transpose()
```

```
Out[71]: array([[ 10,  -5,  30],
                [ -7,   1,   1],
                [  0, 200,  -1],
                [ 20,  40,   4]])
```

Sorting

In [72]:
```
array4 = np.array([1, 0, 2, -3, 6, 8, 4, 7])
array4
```

Out[72]:
```
array([ 1,  0,  2, -3,  6,  8,  4,  7])
```

In [74]:
```
array4.sort()
array4
```

Out[74]:
```
array([-3,  0,  1,  2,  4,  6,  7,  8])
```

In [80]:
```
array5 = np.array([[10, -7, 0, 20],
                   [-5, 1, 200, 40],
                   [30, 1, -1, 4]])
array5
```

Out[80]:
```
array([[ 10,  -7,   0,  20],
       [ -5,   1, 200,  40],
       [ 30,   1,  -1,   4]])
```

In [78]:
```
# default is row-wise sorting

array5.sort()
array5
```

Out[78]:
```
array([[ -7,   0,  10,  20],
       [ -5,   1,  40, 200],
       [ -1,   1,   4,  30]])
```

In [ ]:
```
# axis = 0 means column-wise sorting
```

In [81]:
```
array5.sort(axis=0)
array5
```

Out[81]:
```
array([[ -5,  -7,  -1,   4],
       [ 10,   1,   0,  20],
       [ 30,   1, 200,  40]])
```

In [91]:
```
# Concatenating Arrays

array1 = np.array([[10, 20, 30],[-30, 40, 20]])
array2 = np.zeros((2,3), dtype=array1.dtype)

print(array1)
print()
print(array2)
```

```
[[ 10  20  30]
 [-30  40  20]]

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

In [92]:
```
print(array1.shape)
print()
print(array2.shape)
```

```
(2, 3)

(2, 3)
```

In [93]:
```
np.concatenate((array1,array2), axis=1)
```

Out[93]:
```
array([[ 10,  20,  30,   0,   0,   0],
       [-30,  40,  20,   0,   0,   0]])
```

In [94]:
```
np.concatenate((array1,array2), axis=0)
```

Out[94]:
```
array([[ 10,  20,  30],
       [-30,  40,  20],
       [  0,   0,   0],
       [  0,   0,   0]])
```

Reshaping Arrays

In [96]:
```
array3 = np.arange(10,22)
array3
```

Out[96]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21])
```

In [97]:
```
array3.reshape(3, 4)
```

Out[97]:
```
array([[10, 11, 12, 13],
       [14, 15, 16, 17],
       [18, 19, 20, 21]])
```

In [98]:
```
array3.reshape(2, 6)
```

Out[98]:
```
array([[10, 11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20, 21]])
```

Spliting Arrays

In [100…
```
array4 = np.array([[10, -7, 0, 20],
                   [-5, 1, 200, 40],
                   [30, 1, -1, 4],
                   [1, 2, 0, 4],
                   [0, 1, 0, 2]])
array4
```

Out[100…
```
array([[ 10,  -7,   0,  20],
       [ -5,   1, 200,  40],
       [ 30,   1,  -1,   4],
       [  1,   2,   0,   4],
       [  0,   1,   0,   2]])
```

In [107…
```
first, second, third = np.split(array4,[1, 3])
```

In [108…
```
first
```

Out[108…
```
array([[10, -7,  0, 20]])
```

In [109…
```
second
```

Out[109…
```
array([[ -5,   1, 200,  40],
       [ 30,   1,  -1,   4]])
```

In [110…
```
third
```

Out[110…
```
array([[1, 2, 0, 4],
       [0, 1, 0, 2]])
```

In [111…
```
firstc, secondc, thirdc = np.split(array4,[1,2], axis=1)
```

In [112…
```
firstc
```

Out[112…
```
array([[10],
       [-5],
       [30],
       [ 1],
       [ 0]])
```

In [113…
```
secondc
```

Out[113…
```
array([[-7],
       [ 1],
       [ 1],
       [ 2],
       [ 1]])
```

In [114…
```
thirdc
```

Out[114…
```
array([[  0,  20],
       [200,  40],
       [ -1,   4],
       [  0,   4],
       [  0,   2]])
```

In [118…
```
firsthalf, secondhalf = np.split(array4,2,axis=1)
```

In [119…
```
firsthalf
```

Out[119…
```
array([[10, -7],
       [-5,  1],
       [30,  1],
       [ 1,  2],
       [ 0,  1]])
```

In [120…
```
secondhalf
```

Out[120…
```
array([[  0,  20],
       [200,  40],
       [ -1,   4],
       [  0,   4],
       [  0,   2]])
```

Statistical Operations on Arrays

In [121…
```
# Let us consider two arrays:

arrayA = np.array([1, 0, 2, -3, 6, 8, 4, 7])
arrayB = np.array([[3, 6], [4, 2]])

print(arrayA)
print()
print(arrayB)
```

```
[ 1  0  2 -3  6  8  4  7]

[[3 6]
 [4 2]]
```

1. The max() function finds the maximum element from an array.

```
In [122…
# max element form the whole 1-D array

arrayA.max()
```

```
Out[122…
8
```

```
In [123…
# max element form the whole 2-D array

arrayB.max()
```

```
Out[123…
6
```

```
In [124…
# if axis=1, it gives column wise maximum

arrayB.max(axis = 1)
```

```
Out[124…
array([6, 4])
```

```
In [125…
# if axis=0, it gives row wise maximum

arrayB.max(axis=0)
```

```
Out[125…
array([4, 6])
```

2. The min() function finds the minimum element from an array.

```
In [127…
arrayA.min()
```

```
Out[127…
-3
```

```
In [128…
arrayB.min()
```

```
Out[128…
2
```

```
In [129…
arrayB.min(axis=0)
```

Out[129…

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

3. The sum() function finds the sum of all elements of an array.

In [132…

```
arrayA.sum()
```

Out[132…

```
25
```

In [133…

```
arrayB.sum()
```

Out[133…

```
15
```

In [134…

```
# axis is used to specify the dimension
# on which sum is to be made. Here axis = 1
# means the sum of elements on the first row

arrayB.sum(axis=1)
```

Out[134…

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

4. The mean() function finds the average of elements of the array.

In [135…

```
arrayA.mean()
```

Out[135…

```
3.125
```

In [136…

```
arrayB.mean()
```

Out[136…

```
3.75
```

In [137…

```
# Column average

arrayB.mean(axis = 0)
```

Out[137…

```
array([3.5, 4. ])
```

In [138…

```
arrayB.mean(axis = 1)
```

Out[138…

```
array([4.5, 3. ])
```

5. The std() function is used to find standard deviation of an array of elements.

```
In [139…  arrayA.std()
```

```
Out[139…  3.550968177835448
```

```
In [140…  arrayB.std()
```

```
Out[140…  1.479019945774904
```

```
In [141…  arrayB.std(axis=0)
```

```
Out[141…  array([0.5, 2. ])
```

```
In [142…  arrayB.std(axis=1)
```

```
Out[142…  array([1.5, 1. ])
```

Iteration in the Array

In One Dimensional Array

```
In [15]:  a1 = np.array([1, 2, 3, 4, 5])
```

```
In [16]:  for x in a1:
              print(x)
```

```
1
2
3
4
5
```

```
In [17]:  for i,x in enumerate(a1):
              print(i, x)
```

```
0 1
1 2
2 3
3 4
4 5
```

In Two Dimensional Array

In [22]:
```
a2 = np.array([[1, 2, 3, 4, 5],
               [6, 7, 8, 9, 10]])
```

In [24]:
```
for z in a2:
    print(z)
```

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

In [26]:
```
for i in a2:
    for j in i:
        print(j)
```

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

Loading Array from Filees).

Using NumPy.loadtxt()

Sometimes, we may have data in files and we may need to load that data in an array for processing. numpy.loadtxt() and numpy.genfromtxt() are the two functions that can be used to load data from text files. The most commonly used file type to handle large amount of data is called CSV (Comma Separated Values).

In [150…
```
StudentData = np.loadtxt("data.txt",  skiprows = 1, delimite
StudentData
```

```
Out[150…  array([[ 1, 36, 18, 57],
                 [ 2, 22, 23, 45],
                 [ 3, 43, 51, 37],
                 [ 4, 41, 40, 60],
                 [ 5, 13, 18, 37]])
```

```
In [153…  # To import data into multiple NumPy arrays row wise.
          # Values related to student1 in array stud1, student2 in arra

          stud1, stud2, stud3, stud4, stud5 = np.loadtxt("data.txt", sl
```

```
In [154…  stud1
```

```
Out[154…  array([ 1, 36, 18, 57])
```

```
In [155…  stud2
```

```
Out[155…  array([ 2, 22, 23, 45])
```

```
In [156…  stud3
```

```
Out[156…  array([ 3, 43, 51, 37])
```

```
In [157…  stud4
```

```
Out[157…  array([ 4, 41, 40, 60])
```

```
In [158…  stud5
```

```
Out[158…  array([ 5, 13, 18, 37])
```

```
In [164…  # Import data into multiple arrays column wise.
          # Data in column RollNo will be put in array rollno, data in
          # will be put in array mks1 and so on.

          rollno, mks1, mks2, mks3 = np.loadtxt('MissingData.txt', ski
```

```
In [165…  rollno
```

```
Out[165…  array([1, 2, 3, 4, 5])
```

```
In [166…  mks1
```

```
Out[166…  array([36, 22, 43, 41, 13])
```

In [167…
```
mks2
```
Out[167…
```
array([18, 23, 51, 40, 18])
```
In [168…
```
mks3
```
Out[168…
```
array([57, 45, 37, 60, 37])
```

Using NumPy.genfromtxt()

In [172…
```
DataArray = np.genfromtxt("MissingData.txt", skip_header=1,de
```
In [173…
```
DataArray
```
Out[173…
```
array([[ 1., 36., 18., 57.],
       [ 2., nan, 23., 45.],
       [ 3., 43., 51., nan],
       [ 4., 41., 40., 60.],
       [ 5., 13., 18., 37.]])
```
In [175…
```
dataarray = np.genfromtxt('MissingData.txt',skip_header=1, de
dataarray
```
Out[175…
```
array([[   1,   36,   18,   57],
       [   2, -999,   23,   45],
       [   3,   43,   51, -999],
       [   4,   41,   40,   60],
       [   5,   13,   18,   37]])
```

Saving NumPy Arrays in Files on Disk

The savetxt() function is used to save a NumPy array to a text file

In [176…
```
np.savetxt('testout.txt', dataarray, delimiter=',', fmt='%i'
```