Numpy - **Num**erical **Py**thon

Introduction

NumPy is a Python package. It stands for 'Numerical Python'. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.

Numeric, the ancestor of NumPy, was developed by Jim Hugunin. Another package Numarray was also developed, having some additional functionalities. In **2005**, Travis Oliphant created NumPy package by incorporating the features of Numarray into Numeric package. There are many contributors to this open source project.

Operations using NumPy

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

NumPy is often used along with packages like **SciPy (Scientific Python) and Mat−plotlib (plotting library)**. This combination is widely used as a replacement for MatLab.

Installation on NumPy

Standard Python distribution doesn't come bundled with NumPy module. A lightweight alternative is to install NumPy using popular Python package installer, pip.

```
pip install numpy
```

The best way to enable NumPy is to use an **installable binary package** specific to your operating system.

These binaries contain full **SciPy stack** (inclusive of NumPy, SciPy, matplotlib, IPython, SymPy and nose packages along with core Python).

How to import NumPy

Any time you want to use a package or library in your code, you first need to make it accessible.

```
import numpy as np
```

Difference between a Python list and a NumPy array

NumPy gives you an enormous range of fast and efficient numerically-related options.

While a Python list can contain **different data types (heterogenous)** within a single list, all of the elements in a NumPy array should be **homogenous**.

The mathematical operations that are meant to be performed on arrays wouldn't be possible if the arrays weren't homogenous.

Ndarray Object

The most important object defined in NumPy is an N-dimensional array type called **ndarray**.

It describes the collection of items of the **same type**.

Items in the collection can be accessed using a **zero-based index**.

Every item in an ndarray takes the **same size of block** in the memory.

Each element in ndarray is an object of data-type object (called **dtype**).

The **rank** of the array is the **number of dimensions**.

The **shape** of the array is a tuple of integers giving the **size of the array along each dimension**.

Creating arrays in NumPy

Different functions used for creating ndarray are -

- empty()
- array()
- zeros()
- ones()
- empty()
- arange()
- linspace()
- logspace()

Creating using empty() function

It creates an uninitialized array of specified shape and dtype.

Syntax – `empty(shape, dtype = float)`

- Shape - Shape of an empty array in int or tuple of int
- Dtype - Desired output data type. Optional

```
#Program1 - To demonstrate empty() function
import numpy as np
x = np.empty([3,2], dtype = int)
print (x)
```

```
[[502511173709 420906795117]
 [450971566177 433791696964]
 [446676598892          105]]
```

Note: The elements in an array show random values as they are not initialized.

Creating using zeros() function

Returns a new array of specified size, filled with zeros.

Syntax - `zeros(shape, dtype = float)`

Creating using ones() function

Returns a new array of specified size, filled with ones.

Syntax - `ones(shape, dtype = float)`

```
#Program2 - To demonstrate zeroes() and ones() function
import numpy as np
x = np.zeros(5)
print (x)    # default dtye = float
print('----------------')

x = np.ones(5, dtype = np.int)
print (x)
print('----------------')

x = np.ones([2,2], dtype = int)  # 2X2 array
print (x)
```

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

Creating using array() function

Syntax - `array(object, dtype)`

It creates an ndarray from any object exposing array interface, or from any method that returns an array.

```
#Program 3 - To demonstrate array() function
import numpy as np
x = np.array([1,2,3,4,5],int)
print (x)
print('----------------')

x = np.array([1,2,3,4,5],float)
print (x)
print('----------------')

x = np.array(['a','b','c'])  # For character, no need to specify datatype
print (x)
print('----------------')

x = np.array(['Mumbai','Delhi'],dtype=str)
print (x)
print('----------------')

# It is possible to create from another array

a = np.array((1,2,4,5))  # creating array using tuple object
b = np.array(a)           # creating array using array object
c = a                    # creating array using assignment operator
print(a)
print(b)
print(c)
```

```
[1 2 3 4 5]
----------------
[1. 2. 3. 4. 5.]
----------------
['a' 'b' 'c']
----------------
['Mumbai' 'Delhi']
----------------
[1 2 4 5]
[1 2 4 5]
[1 2 4 5]
```

Creating using arange() function

This function returns an ndarray object containing evenly spaced values within a given range.

Syntax - `arange(start, stop, step, dtype)`

```
#Program 3 - To demonstrate arange() function
import numpy as np
x = np.arange(1,10)              # (start,stop)
print (x)
print('--------------------')

x = np.arange(1,10,2)           # (start,stop, step)
print (x)
print('                    ')
```

```
print( ---------- )

x = np.arange(1,10,2,float)     # (start,stop, step, dtype)
print (x)
print('--------------------')
```

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

Creating using linspace() function

This function is similar to arange() function. In this function, instead of step size, the number of evenly spaced values between the interval is specified.

Syntax - `linspace(start, stop, num, endpoint,dtype)`

- **start** - The starting value of the sequence
- **stop** - The end value of the sequence, included in the sequence if endpoint set to true
- **num** - The number of evenly spaced samples to be generated. Default is 50
- **endpoint** -True by default, hence the stop value is included in the sequence. If false, it is not included

Creating using logspace() function

Returns an ndarray object that contains the numbers that are evenly spaced on a log scale.

Start and stop endpoints of the scale are indices of the base, usually 10.

Syntax - `logspace(start, stop, num, endpoint, base, dtype)`

- **start** - The starting point of the sequence is (base ^ start)
- **stop** - The final value of sequence is (base ^ stop)
- **num** - The number of values between the range. Default is 50
- **endpoint** - If true, stop is the last value in the range
- **base** - Base of log space, default is 10
- **dtype** - Data type of output array. If not given, it depends upon other input arguments.

```
#Program 4 - To demonstrate linspace() and logspace() function
import numpy as np
x = np.linspace(10,30,5)  # step size = (stop - start)/(n-1)
print (x)
print('--------------------')
x = np.linspace(10,30, 5, endpoint = False)   # step size = (stop - start)/n
print (x)
print('--------------------------')
```

```
#Logspace

# default base is 10
x = np.logspace(1.0, 2.0, num = 4)
print (x)
print('----------------------------')

x = np.logspace(1.0, 2.0, num = 4, base =2)  #base is binary
print (x)
```

```
[10. 15. 20. 25. 30.]
--------------------
[10. 14. 18. 22. 26.]
--------------------------
[ 10.          21.5443469   46.41588834 100.        ]
--------------------------
[2.        2.5198421 3.1748021 4.       ]
```

Mathematical Operations

It is possible to perfrom various mathematical operations like addition, subtraction, division, etc. on the element of an array.

Also, the math module functions can also be applied to the elements of the NumPy array.

These operations are performed as **Vectorized Operations** i.e. entire array is processed just like a variable.

Advantages of vectorized operations -

- Vectorized Operations are faster.
- Vectorized Operations are syntactically cleaner i.e provides compact code.

Some Useful Mathematical Functions in NumPY

Function	Meaning
sin(arr)	Calculates sine value of each element in the array
cos(arr)	Calculates cosine value of each element in the array
tan(arr)	Calculates tangent value of each element in the array
arcsin(arr)	Calculates sine inverse value of each element in the array
arcos(arr)	Calculates cosine inverse value of each element in the array
arctan(arr)	Calculates tangent inverse value of each element in the array
log(arr)	Calculates natural logarithmic value of each element in the array
abs(arr)	Calculates absolute value of each element in the array
sqrt(arr)	Calculates square root value of each element in the array
power(arr,n)	Calculates power value of each element in the array when raise to 'n'
exp(arr)	Calculates exponentiation value of each element in the array
sum(arr)	Calculates sum of all elements in the array

Function	Meaning
prod(arr)	Calculates product of all elements in the array
min(arr)	Return smallest element in the array
max(arr)	Return largest element in the array
mean(arr)	Return mean value of all elements in the array
median(arr)	Return median value of all elements in the array
var(arr)	Return variance value of all elements in the array
covar(arr)	Return covariance value of all elements in the array
std(arr)	Gives standard deviation of elements in the array
argmin(arr)	Gives index of the smallest element in the array. Counting start from 0
argmax(arr)	Gives index of the largest element in the array. Counting start from 0
unique(arr)	Gives an array that contains unique elements

```
#Program 5 - To demonstrate Mathematical Operation on NumPy array.
from numpy import *

A = np.array([10,20,30,40])
print('Original array - ',A)

# Arithmetic Operations
print('Arithmetic Operations')
print('After adding 5 - ',A+5)
print('After subtracting  5 - ',A-5)
print('After multiplying with  5 - ',A*5)
print('After dividing by  5 - ',A/5)
print('After modulous with 5 - ',A%5)
print('-------------------------------------')

# Arrays can be use in expressions
print("Array in Expression, the value is ",(A+5)**2-10)
print('-------------------------------------')

#Mathematical Functions
```

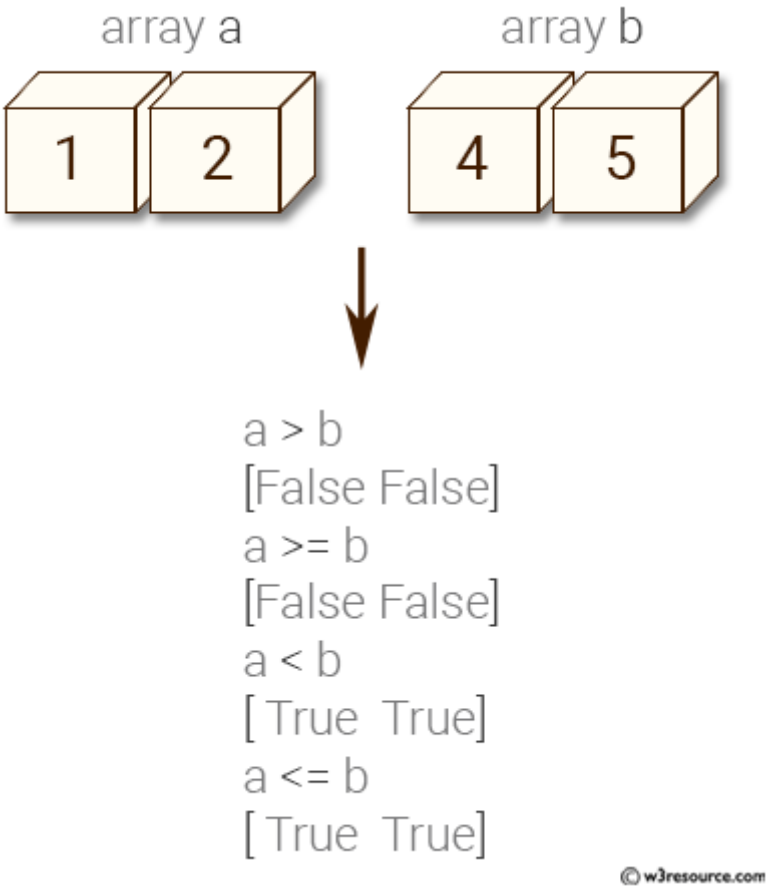
```
#Mathematical Functions
print('Sin values - ',sin(A))
print('Smallest Element - ',min(A))
print('Largest Element - ',max(A))
print('Sum of all elements - ',sum(A))
```

```
Original array -  [10 20 30 40]
Arithmetic Operations
After adding 5 -  [15 25 35 45]
After subtracting  5 -  [ 5 15 25 35]
After multiplying with  5 -  [ 50 100 150 200]
After dividing by  5 -  [2. 4. 6. 8.]
After modulous with 5 -  [0 0 0 0]
---------------------------------------
Array in Expression, the value is  [ 215  615 1215 2015]
---------------------------------------
Sin values -  [-0.54402111  0.91294525 -0.98803162  0.74511316]
Smallest Element -  10
Largest Element -  40
Sum of all elements -  100
```

Comparing Arrays

Relational operators like ==, <, <=, >, >= ,!= are use to compare arrrays of same size.

These operator perform element wise comparison and returns an array with Boolean type values.




```
#Program 6 - To demonstrate Comparing of NumPy array.
from numpy import *
```

```
from numpy import *
a = array([1, 2, 3, 0])
b = array([0, 1, 2, 3])
print("Array a: ",a)
print("Array b: ",b)

# Using relational operator
print("Result of a > b  - ", (a>b))
print("Result of a >= b - ", (a>=b))
print("Result of a < b  - ", (a<b))
print("Result of a <= b - ", (a<=b))
print("Result of a == b - ", (a==b))
print("Result of a != b - ", (a!=b))
print('--------------------------------')

# Using numpy functions
print("Result of a > b  - ", greater(a, b))
print("Result of a >= b - ", greater_equal(a, b))
print("Result of a < b  - ", less(a, b))
print("Result of a <= b - ", less_equal(a, b))
print("Result of a == b - ", equal(a,b))
print("Result of a != b - ", not_equal(a,b))
```

```
Array a:  [1 2 3 0]
Array b:  [0 1 2 3]
Result of a > b  -  [ True  True  True False]
Result of a >= b -  [ True  True  True False]
Result of a < b  -  [False False False  True]
Result of a <= b -  [False False False  True]
Result of a == b -  [False False False False]
Result of a != b -  [ True  True  True  True]
--------------------------------
Result of a > b  -  [ True  True  True False]
Result of a >= b -  [ True  True  True False]
Result of a < b  -  [False False False  True]
Result of a <= b -  [False False False  True]
Result of a == b -  [False False False False]
Result of a != b -  [ True  True  True  True]
```

any() and all() function

any() returns True if at least one element in a NumPy array evaluates to True. **all()** returns True only if all elements in a NumPy array evaluate to True.

```
#Program 7 - To demonstrate any() and all() function of NumPy array.
from numpy import *
a = array([1, 2, 3, 0])
b = array([0, 1, 2, 3])
c = a>b
print("Result of a > b - ",c)

print('Check if any one element is true - ',any(c))
print('Check if all elements are true   - ',all(c))
```

```
if(any(a>b)):
  print('a contains atleast one element greater than those of b')
```

```
    Result of a > b -  [ True  True  True False]
    Check if any one element is true -  True
    Check if all elements are true   -  False
    a contains atleast one element greater than those of b
```

logical_and, logical_or, logical_not, and logical_xor functions

The Python Numpy logical operators and logical functions are to compute truth value using the Truth table, i.,e Boolean True or false.

Python numpy logical functions are logical_and, logical_or, logical_not, and logical_xor.

```
#Program 8 - To demonstrate logical_and, logical_or, logical_not, and logical_xor functior
from numpy import *

A = array([5, 0, 8, 2, 22, 7, 4])
print('Result of logical_and() function - ', logical_and(A > 3, A < 8))
print('Result of logical_or() function - ', logical_or(A > 8, A < 3))
print('Result of logical_not() function - ', logical_not(A))
```

```
    Result of logical_and() function -  [ True False False False False  True  True]
    Result of logical_or() function -  [False  True False  True  True False False]
    Result of logical_not() function -  [False  True False False False False False]
```

where() function

The where function can be used to create a new array based on whether a given condition is TRUE or False.

Syntax -

```
    array = where(condition, expression1, expression2)
```

If condition is true then expression1 is executed else expression2.

```
#Program 9 - To compare elements of two array and retrieve the biggest element
from numpy import *
a = array([10, 20, 30, 40])
b = array([9, 21, 33, 37])
c = where(a>b,a,b)
print(c)
```

```
    [10 21 33 40]
```

nonzero() function

nonzero() function returns an array that contain indexes of the elements which are not equal to

```
#Program 10 - To retrieve the non zero elements
from numpy import *
a = array([10, 20, 0, -1,  30, 0])
c =nonzero(a)

#display indexes of non zero element
print(c)

#display non zero element
print(a[c])
```

```
(array([0, 1, 3, 4]),)
[10 20 -1 30]
```

Aliasing the Arrays

Aliasing is not **copying**. It means giving another name to the existing object. Heance any modification to the alias object will reflect in the existing object and vice versa.

```
#Program 11 - To understand the effect of alias
from numpy import *
a = arange(6)
b = a
print('Original Array  - ',a)
print('Alias Array     - ',b)

b[0] = 111
print('After Modification')
print('Original Array  - ',a)
print('Alias Array     - ',b)
```

```
Original Array  -  [0 1 2 3 4 5]
Alias Array     -  [0 1 2 3 4 5]
After Modification
Original Array  -  [111   1   2   3   4   5]
Alias Array     -  [111   1   2   3   4   5]
```

Viewing and Copying Arrays

The main difference between a copy and a 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 to the copy **will not affect original array**, and any changes made to the original array will not affect the copy.

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.

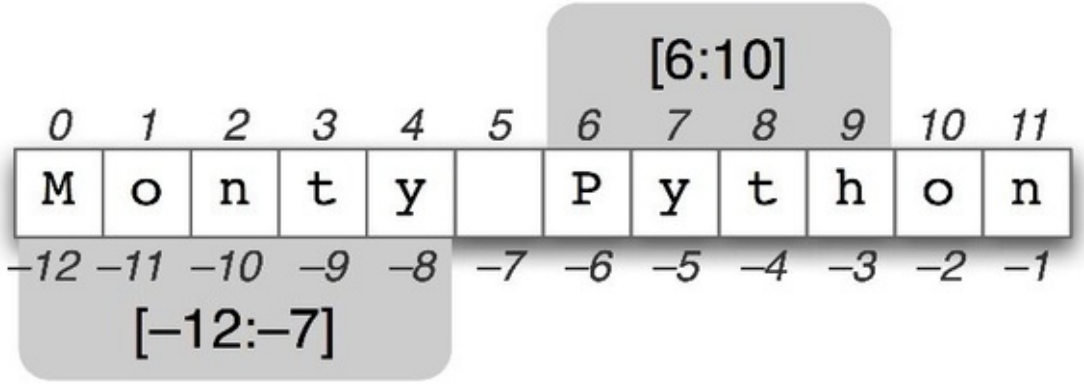
```
#Program 12 - To understand the effect of view() and copy() on an array
from numpy import *

a = array([1, 2, 3, 4, 5])
x = a.copy()
a[0] = 42
print('Effect of copy() function....')
print('Origial Array - ',a)
print('Copied Array  (Deep Copying) - ',x)

a = array([1, 2, 3, 4, 5])
x = a.view()
a[0] = 42
print('Effect of view() function....')
print('Origial Array - ',a)
print('Copied Array (Shallow Copying) - ',x)
```

```
Effect of copy() function....
Origial Array -  [42  2  3  4  5]
Copied Array  (Deep Copying) -  [1 2 3 4 5]
Effect of view() function....
Origial Array -  [42  2  3  4  5]
Copied Array (Shallow Copying) -  [42  2  3  4  5]
```

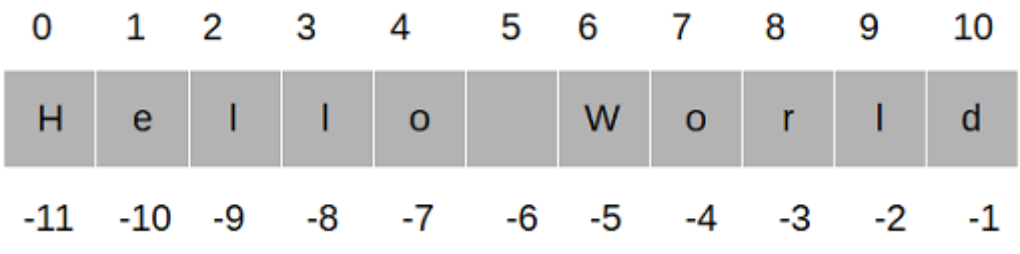
Slicing and Indexing in NumPy array

Slicing refers to extracting a range of element from the array. The format of slicing is

```
arrayname[start:stop:step]
```



Indexing is used to obtain individual elements from an array, but it can also be used to obtain entire rows, columns or planes from multi-dimensional arrays.



```
#Program 13 - To understand slicing and indexing on 1D NumPy array
from numpy import *

a = linspace(1,10,10,dtype=int)
print(a)

b = a[1:6:2]
print (b)

b = a[::]
print (b)


b = a[-2:2:-1]  #retrieve 10 - 2 = 8th element to one element prior to 2nd element in decr
print (b)


b = a[:-2:] #retrieve from  to one element prior to 10-2  = 8th element
print (b)
```

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

Dimensions of Arrays

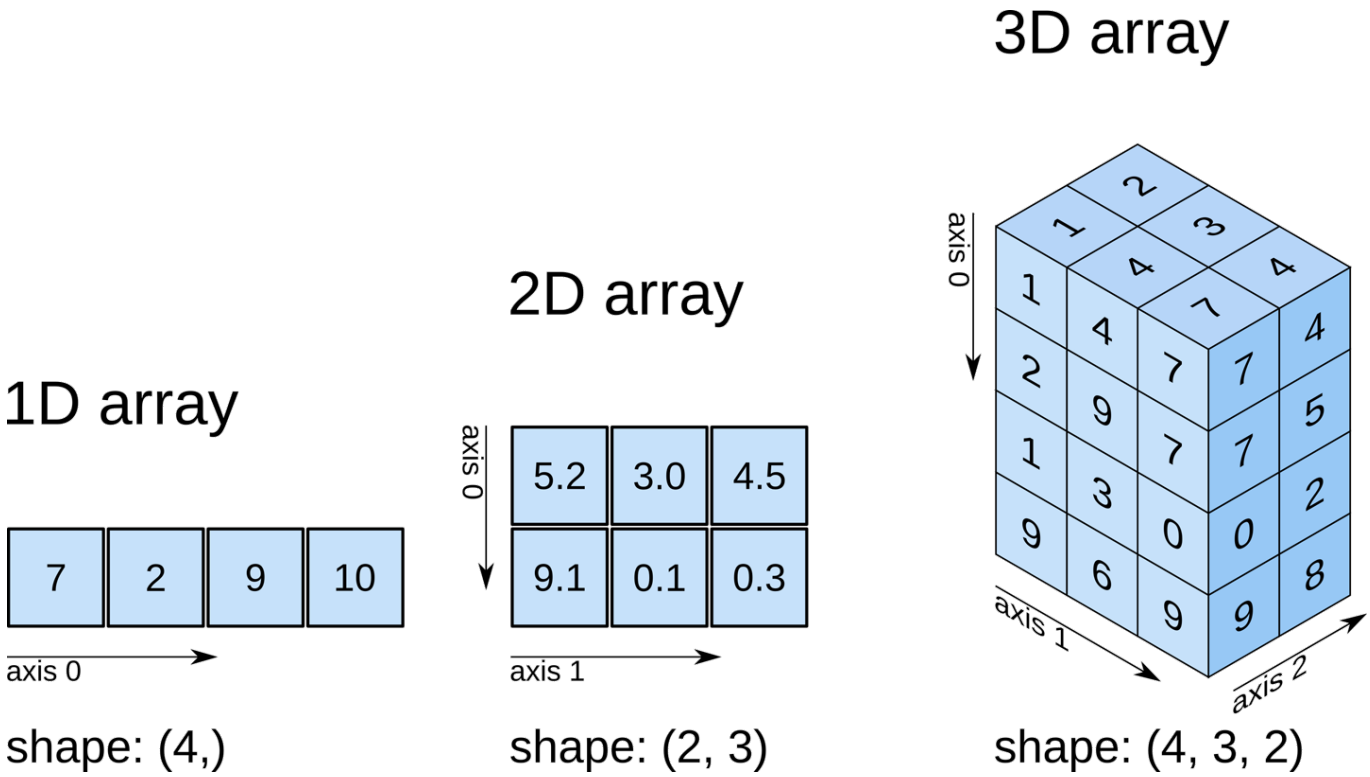
The dimension of an array represents the arrangement of elements in the array.

If the elements are arranged **horizontally**, it is called as **row** and If the elements are arranged **vertically**, it is called as **column**.

When an array contains only one row and one column then it is called single dimensional array or 1D array.

When an array contains more than one row and one column then it is called two dimensional array or 2D array. 2D array canbe refeered as combination of multiple 1D array.

Similarly, 3D array is combination of several 2D array and so on.



Attributes of an Array

NumPy arrays can be thought as **1D** NumPy array as a **list of numbers**, a **2D** NumPy array as a **matrix**, a **3D** NumPy array as a **cube of numbers**, and so on.

- **ndim** attribute gives **dimensions** or axes of a NumPy array. The number of dimension is also referred to as **rank**.
- **shape** attribute gives shape of the array, by creating a tuple listing the number of elements along each dimension. A dimension is called **axis**.
- **size** attribute gives total number of elements in NumPy array.
- **itemsize** attribute gives memory size of the array element in bytes.
- **dtype** attribute gives datatype of the elements in NumPy array.
- **nbytes** attribute gives total number of bytes occuppied by NumPy array.

```
#Program 14 - To crreate different dimensional array and udersatnd different attributes.
from numpy import *

#1D array
one = array([1,2,3,4,5,6,7,8,9],int)
print('1D array   - ',one)
print('Dimesnsion - ',one.ndim)
print('Shape      - ',one.shape)
print('Size       - ',one.size)
print('Itemsize   - ',one.itemsize)
print('Dtype      - ',one.dtype)
print('nbytes     - ',one.nbytes)
```

```
print('------------------------------------')

#2D array
two = array([[1,2,3,4],
             [5,6,7,8]])

print('2D array   - ')
print(two)
print('Dimesnsion - ',two.ndim)
print('Shape      - ',two.shape)
print('Size       - ',two.size)
print('Itemsize   - ',two.itemsize)
print('Dtype      - ',two.dtype)
print('nbytes     - ',two.nbytes)
print('------------------------------------')

#3D array
three = array([[[1,2,3],[4,5,6]],
               [[2,3,4],[6,7,8]]])

print('3D array   - ')
print(three)
print('Dimesnsion - ',three.ndim)
print('Shape      - ',three.shape)
print('Size       - ',three.size)
print('Itemsize   - ',three.itemsize)
print('Dtype      - ',three.dtype)
print('nbytes     - ',three.nbytes)
print('------------------------------------')
```

```
1D array   -  [1 2 3 4 5 6 7 8 9]
Dimesnsion -  1
Shape      -  (9,)
Size       -  9
Itemsize   -  8
Dtype      -  int64
nbytes     -  72
------------------------------------
2D array   -
[[1 2 3 4]
 [5 6 7 8]]
Dimesnsion -  2
Shape      -  (2, 4)
Size       -  8
Itemsize   -  8
Dtype      -  int64
nbytes     -  64
------------------------------------
3D array   -
[[[1 2 3]
  [4 5 6]]

 [[2 3 4]
  [6 7 8]]]
Dimesnsion -  3
Shape      -  (2, 2, 3)
Size       -  12
Itemsize   -  8
```

```
Dtype      -  int64
nbytes     -  96
-----------------------------------
```

Reshape() and Flatten() Method

The **reshape()** method is use to change the shape of the array. The new array should have the same number of elements as in the original array.

The **fatten()** method is use to return the copy of the array collapsed into one dimension.

```
#Program 15 - To understand Reshape() and Flatten() Method on NumPy array.
from numpy import *

a = array([1,2,3,4,5,6,7,8,9],int)
print(a)

a = a.reshape(3,3)
print("\nAfter using reshape() method")
print(a)

a = a.flatten()
print("\nAfter using flatten() method")
print(a)
```

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

After using reshape() method
[[1 2 3]
 [4 5 6]
 [7 8 9]]

After using flatten() method
[1 2 3 4 5 6 7 8 9]
```

Working with Multi-dimensional Arrays

When an array contains more than one row and one column then it is called two dimensional array or 2D array. 2D array can be referred as combination of multiple 1D array. Also known as **matrix**.

Different ways to create mutidimensional array -

- array()
- zeros()
- ones()
- eye()
- reshape()

The array() function

Multidimenional array can be created by passing list of numbers.

If we pass one list of element then 1D array is created whereas if we pass two list of element then 2D array is created and so on.

The ones() and zeros() functions

The **ones()** function is useful to create a 2D array with several rows and columns where all element is initialized to value 1.

Syntax is,

```
    array name = ones((r,c),dtype)
```

The **zeros()** function is useful to create a 2D array with several rows and columns where all element is initialized to value 0.

Syntax is,

```
    array name = zeros((r,c),dtype)
```

The eye() function

The **ones()** function is useful to create a 2D array and fills the element is diagonal position with 1s.

Syntax is,

```
    array name = eye(n,dtype)
```

This will create an array with n rows and n columns. Default datatype is float.

The reshape() function

The **reshape()** function is useful to create a multidimensional array from 1D array.

Syntax is,

```
    array name = reshape(arrayname,(n,r,c))
```

n indicates the number of arrays in the resultant array.

r and c indicates rows and columns

```
#Program 16 - To create multidimensional NumPy array
from numpy import *

# with array()
l1 = [1,2,3,4]; l2 = [5,6,7,8]
```

```
a = array([l1,l2])
print(a,'\n')

# with ones()
a = ones((3,4),int)
print(a,'\n')

# with zeros()
a = zeros((3,4))
print(a,'\n')

# with eye()
a = eye(4,dtype=int)
print(a,'\n')

# with eye()
a = eye(4,3)
print(a,'\n')

# with reshape()
a = arange(12)
print(a,'\n')
b = reshape(a,(4,3))        #2D
c = reshape(a,(3,2,2))      #3D

print(b,'\n')
print(c,'\n')
```

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

[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]

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

[ 0  1  2  3  4  5  6  7  8  9 10 11]

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

[[[ 0  1]
  [ 2  3]]
```

```
[[ 4  5]
 [ 6  7]]

[[ 8  9]
 [10 11]]]
```

Indexing in Multidimensional Array

Index reresents the **location number**.

The individual elements of 2D array can be accessed by specifying the location number of row and column.

```
a[i][j] #represents element in ith row and jth column
```

It is possible to retrieve individual rows separately.

```
a[i] #on 2D array represents complete ith row
```

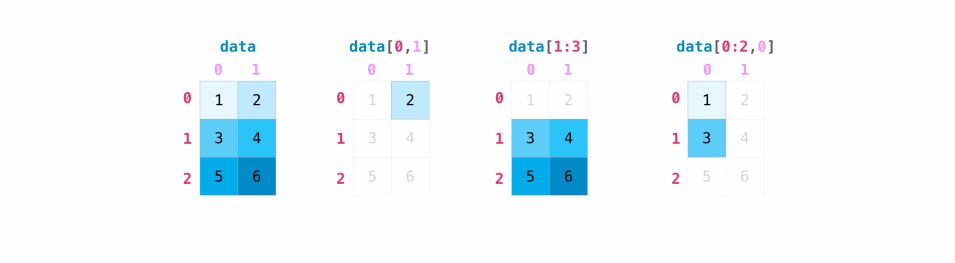
```
#Program 17 - To retrieve the elements from a 2D array and display them in a loop
from numpy import *

a = array([[1, 2, 3, 4],[5, 6, 7, 8],[9, 0, 1, 2]] )

# for displaying only rows
for i in range(len(a)):
  print(a[i])
print()
# for displaying individual elements
for i in range(len(a)):
  for j in range(len(a[i])):
    print(a[i][j],'\t',end='')
  print()
```

```
[1 2 3 4]
[5 6 7 8]
[9 0 1 2]

1       2       3       4
5       6       7       8
9       0       1       2
```

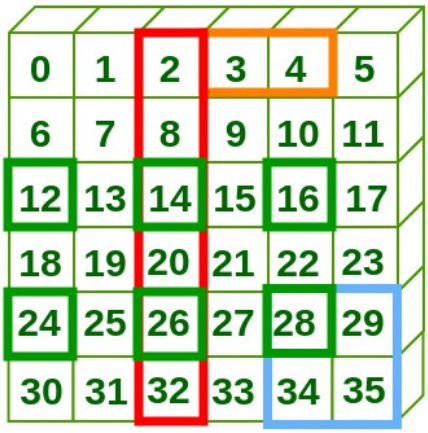
Slicing the Multi-dimensional Array

A slice represents a piece or part of the array. For 2D, Syntax is -

```
arrayname[ start:stop:step , start:stop:step ]
```




```
>>> a[0,3:5]
array( [3,4] )

>>> a[4:, 4:]
array( [ 28, 29],
       [ 34, 35] ] )

>>> a[ :, 2]
array( [2, 8, 14, 20, 26, 32] )

>>> a[2 : : 2, : : 2]
array( [ 12, 14, 16],
       [ 24, 26, 28] ] )
```




```
#Program 18 - To understand effect of slice operation on 2D array.
from numpy import *

a = array([[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12],[13, 14, 15, 16]] )
print(a)

print('\nResult of a[:,:]',a[:,:], sep='\n')    #Entire Array
print('-----------------')

print('\nResult of a[:]',a[:], sep='\n')       #Entire Array
print('-----------------')

print('\nResult of a[0,:]',a[0,:], sep='\n')   #only first row
print('-----------------')

print('\nResult of a[:,0]',a[:,0], sep='\n')   #only first column
print('-----------------')

print('\nResult of a[1:2,1:2]',a[1:2,1:2], sep='\n')   #only one element
print('-----------------')

print('\nResult of a[1:2,1]',a[1:2,1], sep='\n')   #only one element
print('-----------------')

print('\nResult of a[1:3,1:4]',a[1:3,1:4], sep='\n')   # 2 rows and 3 columns
print('-----------------')
```

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

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

Result of a[:,:]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
-----------------

Result of a[:]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
-----------------

Result of a[0,:]
[1 2 3 4]
-----------------

Result of a[:,0]
[ 1  5  9 13]
-----------------

Result of a[1:2,1:2]
[[6]]
-----------------

Result of a[1:2,1]
[6]
-----------------

Result of a[1:3,1:4]
[[ 6  7  8]
 [10 11 12]]
-----------------
```

```
#Program 19 - To understand effect of slice operation on 2D array with negative indexing.
from numpy import *

a = array([[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12],[13, 14, 15, 16]] )
print(a)

n=len(a)
print('\nResult of a[-n,:]',a[-n,:], sep='\n')   #only first row
print('-----------------')

print('\nResult of a[:,-n]',a[:,-n], sep='\n')   #only first column
print('-----------------')

print('\nResult of a[-n+1:-n+2,-n+1:-n+2]',a[-n+1:-n+2,-n+1:-n+2], sep='\n')   #only one e
print('-----------------')

print('\nResult of a[-1:-3,-1:-4]',a[-1:-3,-1:-4], sep='\n')   # 2 rows and 3 columns
print('-----------------')

print('\nResult of a[-1:-3:-1,-1:-4:-1]',a[-1:-3:-1,-1:-4:-1], sep='\n')   # 2 rows and 3
```

```
print('-----------------')

     [[ 1  2  3  4]
      [ 5  6  7  8]
      [ 9 10 11 12]
      [13 14 15 16]]

     Result of a[-n,:]
     [1 2 3 4]
     -----------------

     Result of a[:,-n]
     [ 1  5  9 13]
     -----------------

     Result of a[-n+1:-n+2,-n+1:-n+2]
     [[6]]
     -----------------

     Result of a[-1:-3,-1:-4]
     []
     -----------------

     Result of a[-1:-3:-1,-1:-4:-1]
     [[16 15 14]
      [12 11 10]]
     -----------------
```

Matrices in NumPy

In Mathematics, a matrix reresents a **rectangular array of elements** arranged in rows and columns.

If a matrix has only one row, it is called as **row matrix**. If a matrix has only one column, it is called as **column matrix**. These are nothing but 1D array.

When a matrix has more than 1 row and 1 column, it is called **m x n matrix**, where m represent number of rows and n represents numner of columns.

To work with matrices, numpy provides a special class called **matrix**. In numpy, matrix is a specialized 2D array that retains its 2D nature through operations.

Creating NumPy Matrix

Syntax is

```
    matrixname = matrix(2d array or string)
```

The matrix object receives a 2D array or a string that contains elements which can be converted into a matrix.

```
#Program 20 - To read and create numpy matrix .

from numpy import *
```

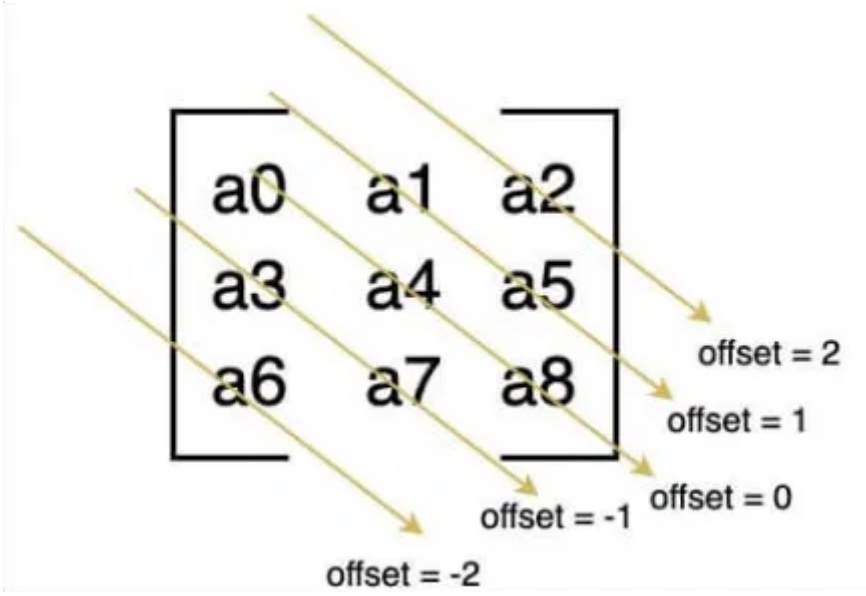
```
a = [[1,2,3],[4,5,6]]   #nested list
m = matrix(a)
print('From nested list,')
print(m)
print('------------------')

a = array([[1,2,3],[4,5,6]],float)   #2d array
m = matrix(a)
print('From 2D array,')
print(m)
print('------------------')

a = '1 2; 3  4; 5 6'   #string
m = matrix(a)
print('From string,')
print(m)
print('------------------')
```

```
From nested list,
[[1 2 3]
 [4 5 6]]
------------------
From 2D array,
[[1. 2. 3.]
 [4. 5. 6.]]
------------------
From string,
[[1 2]
 [3 4]
 [5 6]]
------------------
```

Getting diagonal element of the matrix

diagonal() function is use to retrieve diagonal elements of a matrix. This function returns a **1D array that contains all diagonal elements** of the original array.

Syntax - `a = diagonal(matrixname, offset)`




```
#Program 21 - To print diagonal of a numpy matrix .

from numpy import *

a = [[1,2,3],[4,5,6]]   #nested list
m = matrix(a)
print(m)

d = diagonal(m)
print(d)

d = diagonal(m, offset = 1)
print(d)

d = diagonal(m, offset = 2)
print(d)

d = diagonal(m, offset = -1)
print(d)
```

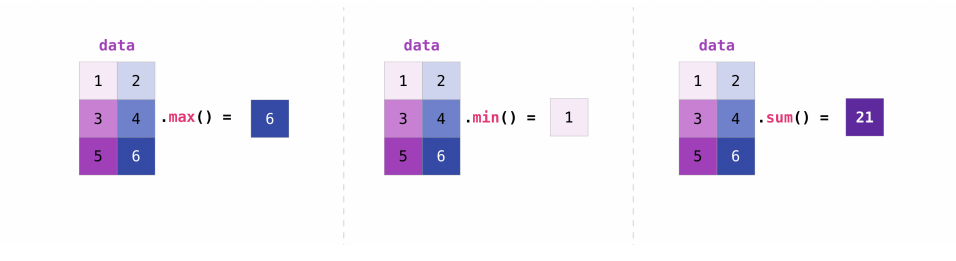
```
[[1 2 3]
 [4 5 6]]
[1 5]
[2 6]
[3]
[4]
```

Finding maximum, minimum, sum, average and product of elements in a matrix

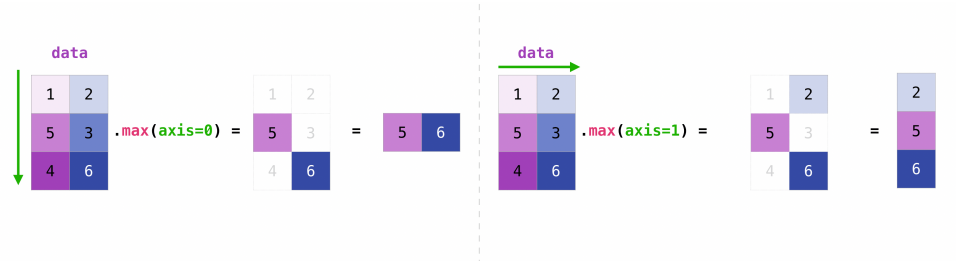
NumPy also performs aggregation functions.

In addition to min, max, and sum, you can easily run mean to get the average, prod to get the result of multiplying the elements together, std to get the standard deviation, and more.

- **max()** is use to find maximum element.
- **min()** is use to find minimum element.
- **sum()** is use to find sum of all element.
- **mean()** is use to find average of all element.
- **prod()** is use to find product of all element.



These methods can also be applied axiswise, where parameter axis need to be specified when making a call. Axis represents dimensions, in 2D, axis = 0 means column wise whereas axis =1 means row wise




```
#Program 22 - To demonstrate aggegate functions on a numpy matrix
from numpy import *

a = [[1,2,3],[4,5,6]]   #nested list
m = matrix(a)
print(m)

print('\nApplying aggreagte functions to entire matrix')
print('Minimum Element         - ',m.min())
print('Maximum Element         - ',m.max())
print('Sum of all Element      - ',m.sum())
print('Product of all Element  - ',m.prod())
print('Average of all Element  - ',m.mean())
print('--------------------------------')

print('\nApplying aggreagte functions Column Wise')
print('Minimum Element         - ',m.min(axis=0))
print('Maximum Element         - ',m.max(axis=0))
print('Sum of all Element      - ',m.sum(axis=0))
print('Product of all Element  - ',m.prod(axis=0))
print('Average of all Element  - ',m.mean(axis=0))
print('--------------------------------')
```

```
print('\nApplying aggreagte functions Row Wise')
print('Minimum Element       -\n ',m.min(axis=1))
print('Maximum Element       -\n ',m.max(axis=1))
print('Sum of all Element     -\n ',m.sum(axis=1))
print('Product of all Element -\n ',m.prod(axis=1))
print('Average of all Element -\n ',m.mean(axis=1))
print('--------------------------------')
```

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

Applying aggreagte functions to entire matrix
Minimum Element         -  1
Maximum Element         -  6
Sum of all Element      -  21
Product of all Element  -  720
Average of all Element  -  3.5
--------------------------------

Applying aggreagte functions Column Wise
Minimum Element         -  [[1 2 3]]
Maximum Element         -  [[4 5 6]]
Sum of all Element      -  [[5 7 9]]
Product of all Element  -  [[ 4 10 18]]
Average of all Element  -  [[2.5 3.5 4.5]]
--------------------------------

Applying aggreagte functions Row Wise
Minimum Element       -
  [[1]
 [4]]
Maximum Element       -
  [[3]
 [6]]
Sum of all Element     -
  [[ 6]
 [15]]
Product of all Element -
  [[  6]
 [120]]
Average of all Element -
  [[2.]
 [5.]]
--------------------------------
```

Sorting the Matix

The sort() functions sorts the matrix elements into ascending order.

Syntax - `result = sort( matrixname, axis)`

If axis = 1, Sorting is performed row-wise else if axis = 0, Sorting is performed column-wise. Default value of axis is set to 1.

```
#Program 23 - To apply sort function on numpy matrix
from numpy import *
```

```
a = [[11,42,37],[41,5,16]]   #nested list
m = matrix(a)
print('Original Matrix')
print(m)
print('---------------')
print('\nAfter row-wise sort..')
b = sort(m)  # row-wise sort
print(b)
print('---------------')
print('\nAfter Column-wise sort..')
c = sort(m, axis=0)  # column-wise sort
print(c)
```

```
Original Matrix
[[11 42 37]
 [41  5 16]]
---------------

After row-wise sort..
[[11 37 42]
 [ 5 16 41]]
---------------

After Column-wise sort..
[[11  5 16]
 [41 42 37]]
```

Transpose os a Matrix

Rewriting matrix rows into columns and vice versa is called Transpose. Thus, rows in the original matrix will become columns in transpose matrix and columns in the original matrix will become rows in transpose matrix.

It means if original matrix has m x n size then transpose will have n x m size.

NumPy rovides two methods to get transpose of a Matrix :

- transpose()
- getT()

```
#Program 24 - To accept a matrix from the keyboard and dislay its transpose matrix
from numpy import *

r, c = [int(a) for a in input('Enter rows and columns - ').split()]
str = input('Enter matrix element - ')

x = matrix(str)
print(x)            # creates 2D Matrix with 1 row and r*c column

x = reshape(matrix(str),(r,c))
print(x)

print('\nThe transpose of the matrix using transpose() function')
```

```
print( \nThe transpose of the matrix using transpose() function )
y = x.transpose()
print(y)

print('\nThe transpose of the matrix using getT() function')
z = x.getT()
print(z)
```

```
Enter rows and columns - 2 3
Enter matrix element - 1 2 3 4 5 6
[[1 2 3 4 5 6]]
[[1 2 3]
 [4 5 6]]

The transpose of the matrix using transpose() function
[[1 4]
 [2 5]
 [3 6]]

The transpose of the matrix using getT() function
[[1 4]
 [2 5]
 [3 6]]
```

Matrix Addition, Subtraction, Divison and Multiplication

Operators i.e `+`, `-`, and `/` acts element wise, whereas `*` performs matrix multiplication according to Mathematics.

```
#Program 25 - To accept two  matrices from the keyboard and perform matrix addition, divis
from numpy import *

r1,c1 = [int(a) for a in input('Enter rows and columns of Matrix 1 - ').split()]
r2,c2 = [int(a) for a in input('Enter rows and columns of Matrix 2- ').split()]

str = input('Enter element of matrix 1- ')
m1 = reshape(matrix(str),(r1,c1))

str = input('Enter element of matrix 2- ')
m2 = reshape(matrix(str),(r2,c2))

print('\nMatrix 1')
print(m1)
print('\nMatrix 2')
print(m2)

print('\nAddition of Matrix 1 and Matrix 2')
print(m1+m2)

print('\nSubtraction of Matrix 1 and Matrix 2')
print(m1-m2)

print('\nDivison of Matrix 1 and Matrix 2')
```

```
print(m1/m2)
```

```
#Program 25 - To accept two  matrices from the keyboard and perform matrix multiplication.
from numpy import *

r1,c1 = [int(a) for a in input('Enter rows and columns of Matrix 1 - ').split()]
r2,c2 = [int(a) for a in input('Enter rows and columns of Matrix 2- ').split()]


if r2!=c1:
  print('\nMatrix multiplication is not possible')
else:
  str = input('Enter element of matrix 1- ')
  m1 = reshape(matrix(str),(r1,c1))
  str = input('Enter element of matrix 2- ')
  m2 = reshape(matrix(str),(r2,c2))
  print('\nMatrix 1')
  print(m1)
  print('\nMatrix 2')
  print(m2)
  print('\nMultiplication of Matrix 1 and Matrix 2')
  print(m1*m2)
```