Experiment 05

Write a python program to understand Basic Array operations on 1-D and 2D and Multidimensional arrays using Array, NumPy.
Also explain the attributes of the array function with examples and demonstrate the concept of matrix creation using matrix and perform different matrix operations. Also show the different ways to create an array and slow slicing operator on 2D array.

Roll No.	61
Name	V Krishnasubramaniam
Class	D10-A
Subject	Python Lab
LO Mapped	LO1: Understand the structure, syntax, and semantics of the Python language LO2: Interpret advanced data types and functions in python

Aim:

Write a python program to understand Basic Array operations on 1-D and 2D and Multidimensional arrays using Array, NumPy.
Also explain the attributes of the array function with examples and demonstrate the concept of matrix creation using matrix and perform different matrix operations.
Also show the different ways to create an array and slow slicing operator on 2D array

Introduction:

Array in Python

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

Array can be handled in Python by a module named array. They can be useful when we have to manipulate only specific data type values. A user can treat lists as arrays. However, users cannot constrain the type of elements stored in a list. If you create arrays using the array module, all elements of the array must be of the same type.

Arrays in Python can be created by importing array module. There are two ways of importing the array module into the program. The first way is to import entire array module using import statement as:

```
import array
a = array.array('i', [4, 6, 2, 9])
```

Here the first 'array' represents the module array and the next 'array' represents the class name for which the object is created. The second way of importing is to import all classes, objects, etc, from the array module into the program.

```
from array import *
a = array('i', [4, 6, 2, 9])
```

Some of the data types are mentioned below which will help in creating an array of different data types.

Type Code	**C Type**	**Minimum size in bytes**
'b'	signed char	1
'B'	unsigned char	1
'i'	signed integer	2
'I'	unsigned integer	2
'l'	signed long	4
'L'	unsigned long	4

‘f’	float	4
‘d’	double	8
‘u’	unicode character	2

To traverse through an array, we can use a for loop to access each element of an array.

Example:
```
from array import *
array1 = array('i', [10,20,30,40,50])
for x in array1:
        print(x)
```

Output:
```
10
20
30
40
50
```

In order to access the array items, refer to the index number. Use the index operator [ ] to access an item in an array. The index must be an integer.

Example:
```
import array as arr
a = arr.array('i', [1, 2, 3, 4, 5, 6])
print("Access element is: ", a[0])
print("Access element is: ", a[3])
```

Output:
```
Access element is: 1
Access element is: 4
```

Elements can be added to the Array by using built-in insert() function. Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of the array. append() can also be used to add the value mentioned in its arguments at the end of the array.

Example:
```
import array as arr
a = arr.array('i', [1, 2, 3])
print ("Array before insertion : ", end =" ")
for i in range (0, 3):
        print (a[i], end =" ")
print()
a.insert(1, 4)
print ("Array after insertion : ", end =" ")
```

```
for i in (a):
        print (i, end =" ")
```

Output:

```
Array before insertion: 123
Array after insertion: 1423
```

Deletion of elements: Elements can be removed from the array by using built-in remove() function but an Error arises if element doesn’t exist in the set. Remove() method only removes one element at a time, to remove range of elements, iterator is used.

Example:

```
import array
arr = array.array('i', [1, 2, 3, 1, 5])
print (“New created array : ", end ="")
for i in range (0, 5):
        print (arr[i], end =" ")
print ("\r")
arr.remove(1)
print ("Array after removing : ", end ="")
for i in range (0, 3):
        print (arr[i], end =" ")
```

Output:

```
New created array: 12315
Array after removing: 2315
```

Pop() function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the pop() method.

Slicing of an array:

In Python arrays, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use slice() operation. Slice operation is performed on an array with the use of colon(:).

Example:

```
import array as arr
l = [1, 2, 3, 4, 5, 6, 7]
a = arr.array('i', l)
print("Initial Array: ")
for i in (a):
        print(i, end =" ")
Sliced_array = a[0:2]
print("\nSlicing elements 0-2: ")
print(Sliced_array)
Sliced_array = a[5:]
```

```
print("\nElements sliced from 2nd to end: ")
print(Sliced_array)
```

Output:

```
Initial array: 1234567
Slicing elements 0-2: 12
Elements sliced from 4th to end: 567
```

To print elements from beginning to a range use [:Index], to print elements from end use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print whole List with the use of slicing operation, use [:]. Further, to print the whole array in reverse order, use [::-1].

In order to search an element in the array, we use a python in-built index() method. This function returns the index of the first occurrence of value mentioned in arguments.

Example:

```
import array
arr = array.array('i', [1, 2, 3, 2])
print ("New created array is: ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
print ("\r")
print ("The index of 1st occurrence of 2 is: ", end ="")
print (arr.index(2))
```

Output:

```
New created array is: 1232
The index of 1st occurrence of 2 is: 1
```

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

Example:

```
import array
arr = array.array('i', [1, 2, 3])
print ("Array before updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
print ("\r")
arr[2] = 6
print("Array after updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
```

Output:

```
Array before updation: 123
Array after updation: 126
```

Two-dimensional Array in Python

It is an array within an array. In this type of array the position of a data element is referred to by two indices instead of one. So it represents a table with rows and columns of data.

In the below example of a two dimensional array, observe that each array element itself is also an array:

T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]

The data elements in two dimensional arrays can be accessed using two indices. One index referring to the main or parent array and another index referring to the position of the data element in the inner array. If we mention only one index then the entire inner array is printed for that index position.

Example:
```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5]]
print(T[0])
print(T[1][2])
```

Output:
```
[11, 12, 5, 2]
10
```

We can insert new data elements at specific positions by using the insert() method and specifying the index.

Example:
```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5]]
T.insert(2, [0,5,11,13,6])
for r in T:
    for c in r:
        print(c,end = " ")
    print()
```

Output:
```
11 12 5 2
15 6 10
0 5 11 13 6
10 8 12 15
```

We can update the entire inner array or some specific data elements of the inner array by reassigning the values using the array index.

Example:
```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5]]
```

```
T[2] = [11,9]
T[0][3] = 7
for r in T:
        for c in r:
                print(c,end = " ")
        print()
```

Output:

```
11 2 5 7
15 6 10
11 9
```

We can delete the entire inner array or some specific data elements of the inner array by reassigning the values using the del() method with index. But in case you need to remove specific data elements in one of the inner arrays, then use the update process described above.

Example:

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5]]
del T[1]
for r in T:
        for c in r:
                print(c,end = " ")
        print()
```

Output:

```
11 12 5 2
10 8 12 5
```

Multi-dimensional Array in Python

These arrays represent more than one row and more than one column of elements. It is called an array of arrays. Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions. We can create a multidimensional array with each element value as 0.

Example:

```
array = [ [0] * c ] * r ]
array = [ [0] * c for i in range(r) ]
```

Numpy

Numpy is a package that contains several classes, functions, variables, etc. to deal with scientific calculations in Python. Numpy is useful to create and also process single and multidimensional arrays. Arrays which are created with numpy are called n-dimensional arrays where n can be any integer. To work with numpy, we should first import numpy modules into the programs.

Example:

```
import numpy
```

```
arr = numpy.array([10, 20, 30, 40, 50])
```

Creation of Array

Creating arrays in numpy can be done in several ways:

Using array() function: We can call array() function to create an array. When we create an array, we can specify the datatype of the elements either as ‘int’ or ‘float’.

Example:
```
from numpy import *
arr = array([‘a’, ‘b’, ‘c’, ‘d’])
print(arr)
```

Output:
```
[‘a’, ‘b’, ‘c’, ‘d’]
```

Using linspace() function: The linspace() function is used to create an array with evenly spaced points between a starting point and ending point. The form of the linspace function is

Example:
```
linspace(start, stop, n)
a = linspace(0, 10, 5)
```

Start represents the starting element and stop represents the ending element. N is an integer that represents the number of parts the element should be divided. If n is omitted, then it is taken 50.

Example:
```
from numpy import *
arr = linspace(0, 10, 5)
print(‘arr = ‘,arr)
```

Output:
```
[   0.    2.5    5.    7.5    10.    ]
```

Using logspace() function: The logspace is similar to the linspace function. The linspace function produces the evenly spaced points . Similarly, logspace() produces evenly spaced points on a logarithmic spaced scale. The form of logspace function is

Example:
```
logspace(start, stop, n)
a = linspace(1, 4, 5)
```

The logspace function starts at the value which starts at the power of 10 and ends at a value which stops at power of 10. If n is not specified then the value is taken as 50.

Example:
```
from numpy import *
arr = logspace(1, 4, 5)
n = len(arr)
for i in range(n)
```

```
        print(‘%.1f‘ % a[i] ,end = ‘ ’)
```

Output:

```
10.0   56.2   316.2   1778.3   10000.3
```

Using arrange() function: The arrange() function is the same as range() function in Python. The form of arrange function is:

1. arrange(start)
2. arrange(start, stop)
3. arrange(start, stop, stepsize)

Example:

```
a = arrange(2)
a = arrange(2, 11)
a = arrange(2, 11, 2)
```

The arrange function creates an array with a group of elements from start to one element prior to stop in steps of step size. If the step size is omitted, then it takes as 1. If the start is omitted, then it takes as 0.

Example:

```
from numpy import *
arr = arrange(2, 11, 2)
print(arr)
```

Output:

```
[2   4   6   8   10]
```

Using zeros() and ones() function: We can use the zeroes() function to create an array with all zeros. The ones() function is useful to create an array with all 1s. The form is:

```
zeros(n, datatype)
ones(n, datatype)
```

Here n represents the number of elements. We can eliminate the datatype argument. If we do not specify the datatype then the default data type used by numpy is float.

Example:

```
zeros(5, int)
ones(2, int)
```

Example:

```
from numpy import *
a = zeros(5, int)
print(a)
b = ones(2, int)
print(b)
```

Output:

```
[0, 0, 0, 0, 0]
[1, 1]
```

Mathematical operations on Arrays

It is possible to perform various mathematical operations like addition, multiplication, subtraction, etc. on the elements of an array. These kinds of operations are also called vectorized operations since the entire array is processed just like a variable.

<u>Example:</u>

```
import numpy as np
arr1 = np.arange(4, dtype = np.float_).reshape(2, 2)
print('First array:')
print(arr1)
print('\nSecond array:')
arr2 = np.array([12, 12])
print(arr2)
print('\nAdding the two arrays:')
print(np.add(arr1, arr2))
print('\nSubtracting the two arrays:')
print(np.subtract(arr1, arr2))
```

<u>Output:</u>

```
First array:
[[0.    1.]
[2.    3.]]
Second array:
[12, 12]
Adding the two arrays:
[[12.    13.]
[14.    15.]]
Subtracting the two arrays:
[[-12.    -11.]
[-10.    -9.]]
```

Comparing two numpy arrays determines whether they are equivalent by checking if every element at each corresponding index are the same. We generally use the == operator to compare two NumPy arrays to generate a new array object. Call ndarray.all() with the new array object as ndarray to return True if the two NumPy arrays are equivalent.

<u>Example:</u>

```
import numpy as np
an_array = np.array([[1, 2], [3, 4]])
another_array = np.array([[1, 2], [3, 4]])
comparison = an_array == another_array
equal_arrays = comparison.all()
print(equal_arrays)
```

Output:
```
True
```

We can also use greater than, less than and equal to operators to compare. To understand, have a look at the code below.

Example:
```
import numpy as np
a = np.array([101, 99, 87])
b = np.array([897, 97, 111])
print("Array a: ", a)
print("Array b: ", b)
print("a > b")
print(np.greater(a, b))
print("a >= b")
print(np.greater_equal(a, b))
```

Output:
```
Array a: [101 99 87]
Array b: [897 97 111]
a>b
[False True False]
a>=b
[False True False]
```

We can create another array that is the same as an existing array. This method creates a copy of an existing array such that the new array will also contain the same elements found in the existing array.

Example:
```
import numpy as np
x = np.array([[0, 1, ], [2, 3]])
print("x is:\n", x)
y = x.copy()
x.fill(1)
print("\n Now x is : \n", x)
print("\n y is: \n", y)
```

Output:
```
x is:
[[0     1]
[2      3]]
Now x is:
[[1     1]
[1      1]]
y is:
[[0     1]
[2        3]]
```

Creating multidimensional arrays in numpy can be done in several ways:

Using array() function: We can call array() function to create an array. When we create a multidimensional array, we need to pass two lists of elements.

<u>Example:</u>

```
from numpy import *
arr = array([1, 2, 3, 4], [5, 6, 7, 8])
print(arr)
```

<u>Output:</u>

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

Using zeros() and ones() function: We can use the zeroes() function to create a multidimensional array with all zeros. The ones() function is useful to create a multidimensional array with all 1s. The form is

```
zeros((r, c), datatype)
ones((r, c), datatype)
```

<u>Example:</u>

```
zeros((2, 4), int)
ones((2, 4), int)
```

Here r represents the number of rows and c represents the number of columns. We can eliminate the datatype argument. If we do not specify the datatype then the default data type used by numpy is float.

<u>Example:</u>

```
from numpy import *
a = zeros((2, 4), int)
print(a)
b = ones((2, 4), int)
print(b)
```

<u>Output:</u>

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

Using eye() function: The eye() function creates a multidimensional array and fills the elements in the diagonals with 1s. The form of the function is: eye(n, datatype)

<u>Example:</u>

```
eye(3, int)
```

This will create an array with n rows and n columns. The default data type is float.

Example:

```
from numpy import *
a = eye(3, int)
print(a)
```

Output:

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

Using reshape() function: The reshape() function shapes an array without changing data of array. The form of the function is: reshape(arrayname, (n, r, c))

Example:

```
reshape(arr, (2, 3))
```

Here arrayname represents the name of an array whose elements to be converted. N indicates the number of arrays in the resultant array. R and C indicate the number of rows and columns, respectively.

Example:

```
from numpy import *
a = array([1, 2, 3, 4, 5, 6])
reshape(a, (2, 3))
print(a)
```

Output:

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

In Python arrays, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use slice() operation. Slice operation is performed on an array with the use of colon(:)

Example:

```
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print a
print 'Slice the array from the index a[1:]'
print a[1:]
```

Output:

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

Slicing can also include ellipsis (…) to make a selection tuple of the same length as the dimension of an array. If ellipsis is used at the row position, it will return an n-dimensional array consisting of items in rows.

To index a multidimensional array you can use reshape() method along with arange(). Inside reshape() the parameters should be the multiple of the arange() parameter.

<u>Example:</u>

```
import numpy as np
arr_m = np.arange(12).reshape(2, 2, 3)
print(arr_m)
```

<u>Output:</u>

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

You can also index with a slicing operation similar to a single dimension array.

<u>Example:</u>

```
import numpy as np
arr_m = np.arange(12).reshape(2, 2, 3)
print(arr_m[0:3])
print()
print(arr_m[1:5:2,::3])
```

<u>Output:</u>

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

Built-in functions related to array

Function	**Description**
abs, fabs	Compute the absolute value element-wise for integer, floating point, or complex values. Use fabs as a faster alternative for non-complex-valued data
sqrt	Compute the square root of each element. Equivalent to arr ** 0.5
square	Compute the square of each element. Equivalent to arr ** 2
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element

floor	Compute the floor of each element, i.e. the largest integer less than or equal to each element
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise. Equivalent to -arr.
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum. fmax ignores NaN
minimum, fmin	Element-wise minimum. fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators >, >=, <, <=, ==, !=
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation. Equivalent to infix operators & |, ^
sum	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
mean	Arithmetic mean. Zero-length arrays have NaN mean.
std, var	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
min, max	Minimum and maximum.
argmin, argmax	Indices of minimum and maximum elements, respectively.
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1
unique(x)	Compute the sorted, unique elements in x
intersect1d(x, y)	Compute the sorted, common elements in x and y
union1d(x, y)	Compute the sorted union of elements
in1d(x, y)	Compute a boolean array indicating whether each element of x is contained in y
setdiff1d(x, y)	Set difference, elements in x that are not in y

setxor1d(x, y)	Set symmetric differences; elements that are in either of the arrays, but not both
diag	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
dot	Matrix multiplication
trace	Compute the sum of the diagonal elements
det	Compute the matrix determinant
eig	Compute the eigenvalues and eigenvectors of a square matrix
inv	Compute the inverse of a square matrix
pinv	Compute the Moore-Penrose pseudo-inverse inverse of a matrix
qr	Compute the QR decomposition
svd	Compute the singular value decomposition (SVD)
solve	Solve the linear system Ax = b for x, where A is a square matrix
lstsq	Compute the least-squares solution to Ax = b
seed	Seed the random number generator
permutation	Return a random permutation of a sequence, or return a permuted range
shuffle	Randomly permute a sequence in place
rand	Draw samples from a uniform distribution
randint	Draw random integers from a given low-to-high range
randn	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
binomial	Draw samples from a binomial distribution
normal	Draw samples from a normal (Gaussian) distribution
beta	Draw samples from a beta distribution
chisquare	Draw samples from a chi-square distribution
gamma	Draw samples from a gamma distribution
uniform	Draw samples from a uniform [0, 1) distribution

Matrices in Python

A Python matrix is a specialized two-dimensional rectangular array of data stored in rows and columns. The data in a matrix can be numbers, strings, expressions, symbols, etc. Matrix is one of the important data structures that can be used in mathematical and scientific calculations. A lot of operations can be done on a matrix-like addition, subtraction, multiplication, etc. The python matrix makes use of arrays, and the same can be implemented in 2 ways:

1. Create a Python Matrix using the nested list data type

   In Python, the arrays are represented using the list data type. So now will make use of the list to create a python matrix.

   Syntax:

```
myMatrix = [[Row1],            [Row2],            [Row3], … , [RowN]]
```

Example:

```
M1 = [[8, 14, -6],           [12,7,4],           [-11,3,21]]
#To print the matrix
print(M1)
```

Output:

```
[[8, 14, -6], [12, 7, 4], [-11, 3, 21]]
```

2. Create Python Matrix using matrix from Python Numpy package

NumPy is a package for scientific computing which has support for a powerful N-dimensional array object. To make use of Numpy in your code, you have to import it using the command: import NumPy as np. This class returns a matrix from a string of data or array-like object. Matrix obtained is a specialised 2D array.

Syntax:

```
numpy.matrix(data, dtype = None) :
```

Example:

```
import numpy as np
a = np.matrix('1 2; 3 4')
print("Via string input : \n", a, "\n\n")
b = np.matrix([[5, 6, 7], [4, 6]])
print("Via array-like input : \n", b)
```

Output:

```
Via string input :
 [[1 2]
 [3 4]]
Via array-like input :
 [[[5, 6, 7]
[4, 6]]]
```

To get the order of the matrix we use shape attribute.

Examples:

```
l = [[1, 2, 3], [3, 6, 9], [2, 4, 6]]
a = numpy.matrix(l)
print(a)
```

Output:

```
[[1 2 3]
 [3 6 9]
 [2 4 6]]
```

```
print(a[0])
```

Output:

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

Matrix Manipulation in Python

We can retrieve the diagonal elements of a matrix using the diagonal function.

Example:
```
m = matrix('1 2 3; 4 5 6; 7 8 9')
d = diagonal(m)
print(d)
```

Output:
```
[1 5 9]
```

Maximum and minimum elements in matrix can be found by using max() and min() function respectively. Similarly sum and average of all elements can also be found using sum() and mean() methods

Example:
```
m = matrix('1 2 3; 4 5 6; 7 8 9')
max = m.max()
min = m.min()
sum = m.sum()
avg = m.mean()
print(max,min,sum,avg)
```

Output:
```
9 1 45 5.0
```

Algebraic matrix multiplication can be performed.

Example:
```
m = numpy.matrix([[1,2],[3,4]])
print(m*m)
```

Output:
```
matrix([[ 7, 10], [15, 22]])
```

n-th order algebraic matrix power can also be found.

Example:
```
print(m**3)
```

Output:
```
matrix([[ 37, 54], [ 81, 118]])
```

numpy provides sort() fuunction to sirt the matrix elements into ascending order.

Example:
```
m = matrix('5 4 2; 2 7 0')
a = sort(m)
print(a)
b = sort(m, axis=0)
```

```
print(b)
```

Output:

```
[[1 4 5]
 [0 2 7]]
[[2 4 0]
 [5 7 1]]
```

Transpose of a matrix can be found using transpose() and getT() methods in numpy.

Example:

```
m = matrix('1 2 3; 4 5 6; 7 8 9')
print(m)
t = m.transpose()
print(t)
```

Output:

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

We can use arithmetic operators like +, - and / to perform addition, subtraction and division operations on 2 matrices.

Example:

```
a = matrix('1 2 3; 4 5 6')
b= matrix('2 2 2; 1 –2 2')
c=a+b
print(c)
d=a/b
print(d)
```

Output:

```
[[3 4 5]
 [5 4 8]]
[[0.5 1. 1.5]
 [4. -5. 3.]
```

Two matrices can also be multiplied

Example:

```
a = matrix('1 2 3; 4 5 6')
b= matrix('1 0; 0 2; 1 -1')
c=a*b
print(c)
```

Output:
```
[[4 1]
 [10 4]]
```

Results

Basic Array Operations:

```
>>> from array import *
>>> arr = array('i',[1,2,3,4,5,6,7,8,9,10])
>>> for x in arr: print(x)
...
1
2
3
4
5
6
7
8
9
10
```

```
>>> import array as arr
>>> a = arr.array('i', [10, 20, 30, 40, 50])
>>> print("Accessing element at index 0 is: ", a[0])
Accessing element at index 0 is:  10
>>> print("Accessing element at index 3 is: ", a[3])
Accessing element at index 3 is:  40
```

```
>>> a = arr.array('i',[10,20,30])
>>> print(a)
array('i', [10, 20, 30])
>>> a.insert(1,15)
>>> print(a)
array('i', [10, 15, 20, 30])
```

```
>>> a.append(99)
>>> print(a)
array('i', [10, 15, 20, 30, 99])
>>> a.remove(10)
>>> print(a)
array('i', [15, 20, 30, 99])
```

```
>>> a.pop()
99
>>> print(a)
array('i', [15, 20, 30])
```

```
>>> a[1] = 200
>>> print(a)
array('i', [15, 200, 30])
>>> a.index(30)
2
```

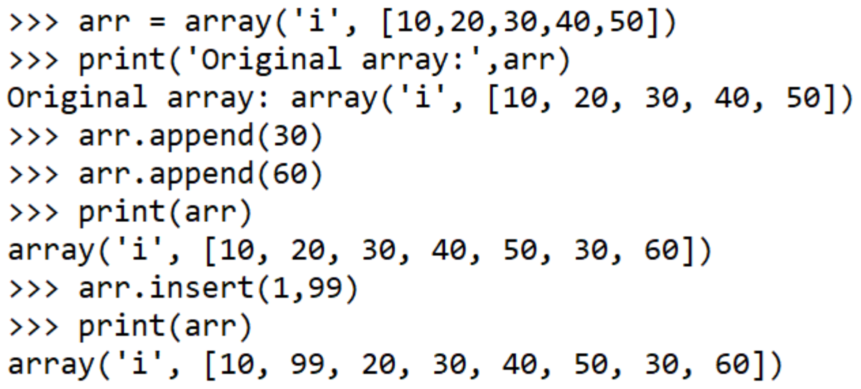
```
>>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a = arr.array('i', l)
>>> a[3:8]
array('i', [4, 5, 6, 7, 8])
>>> a[5:]
array('i', [6, 7, 8, 9, 10])
>>> a[:]
array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Operations on 2D Arrays:

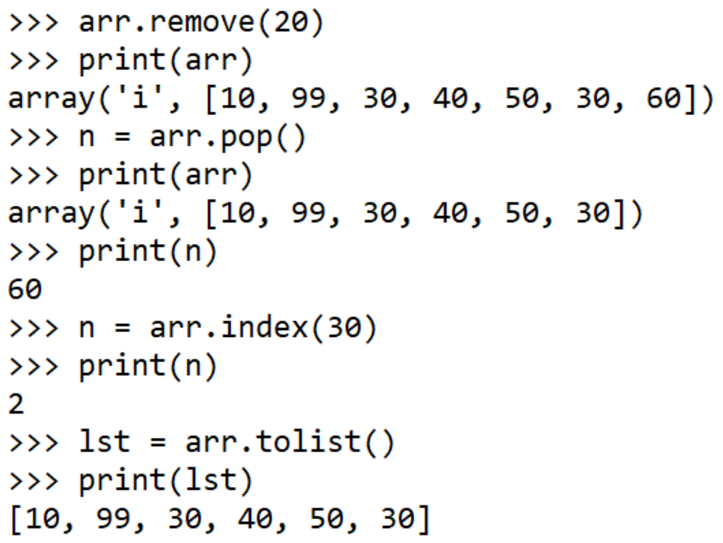
```
>>> from array import *
>>> T = [[11, 12, 5, 2], [15, 6, 10], [10, 8, 12, 5]]
>>> print(T[0])
[11, 12, 5, 2]
>>>
>>> print(T[1][2])
10
```

```
>>> T.insert(2, [0,5,11,13,6])
>>> print(T)
[[11, 12, 5, 2], [15, 6, 10], [0, 5, 11, 13, 6], [10, 8, 12, 5]]
>>> del T[1]
>>> print(T)
[[11, 12, 5, 2], [0, 5, 11, 13, 6], [10, 8, 12, 5]]
>>> T[2] = [11,9]
>>>
>>> T[0][3] = 7
>>> print(T)
[[11, 12, 5, 7], [0, 5, 11, 13, 6], [11, 9]]
```

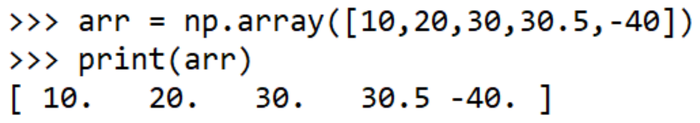
Built-in Functions in NumPy array:



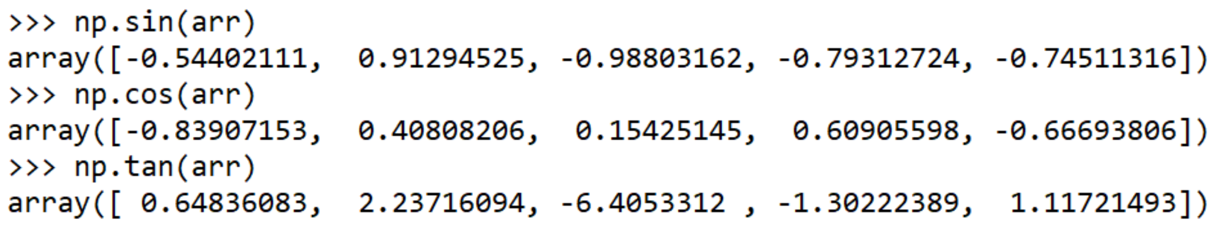
```
>>> arr = array('i', [10,20,30,40,50])
>>> print('Original array:',arr)
Original array: array('i', [10, 20, 30, 40, 50])
>>> arr.append(30)
>>> arr.append(60)
>>> print(arr)
array('i', [10, 20, 30, 40, 50, 30, 60])
>>> arr.insert(1,99)
>>> print(arr)
array('i', [10, 99, 20, 30, 40, 50, 30, 60])
```



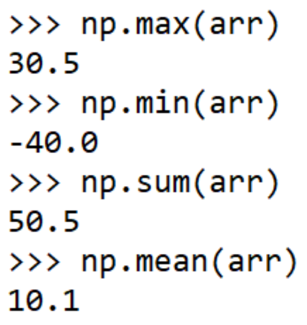
```
>>> arr.remove(20)
>>> print(arr)
array('i', [10, 99, 30, 40, 50, 30, 60])
>>> n = arr.pop()
>>> print(arr)
array('i', [10, 99, 30, 40, 50, 30])
>>> print(n)
60
>>> n = arr.index(30)
>>> print(n)
2
>>> lst = arr.tolist()
>>> print(lst)
[10, 99, 30, 40, 50, 30]
```



```
>>> arr = np.array([10,20,30,30.5,-40])
>>> print(arr)
[ 10.   20.   30.   30.5 -40. ]
```



```
>>> np.sin(arr)
array([-0.54402111,  0.91294525, -0.98803162, -0.79312724, -0.74511316])
>>> np.cos(arr)
array([-0.83907153,  0.40808206,  0.15425145,  0.60905598, -0.66693806])
>>> np.tan(arr)
array([ 0.64836083,  2.23716094, -6.4053312 , -1.30222389,  1.11721493])
```



```
>>> np.max(arr)
30.5
>>> np.min(arr)
-40.0
>>> np.sum(arr)
50.5
>>> np.mean(arr)
10.1
```

```
>>> a = np.array([1,2,3],int)
>>> b = np.array([0,2,3],int)
```

```
>>> c = np.logical_and(a>0,a<4)
>>> print(c)
[ True  True  True]
>>> c = np.logical_or(b>=0, b==1)
>>> print(c)
[ True  True  True]
>>> c = np.logical_not(b)
>>> print(c)
[ True False False]
```

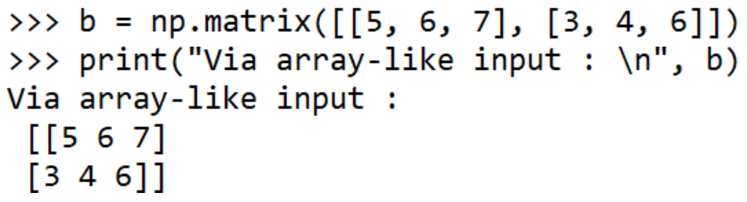
```
>>> a = np.array([1,2,0,-1,0,6],int)
>>> c = np.nonzero(a)
>>> print(a[c])
[ 1  2 -1  6]
```

```
>>> a = np.arange(1,6)
>>> print(a)
[1 2 3 4 5]
>>> b = a.copy()
>>> b[0] = 99
>>> print(b)
[99  2  3  4  5]
```

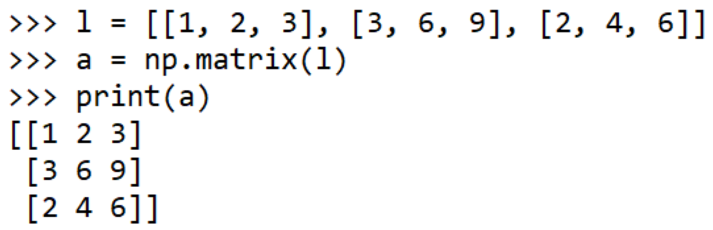
```
>>> a = np.array([[1,2],[2,3],[3,4],[4,5]])
>>> print(a)
[[1 2]
 [2 3]
 [3 4]
 [4 5]]
>>> print(a.flatten())
[1 2 2 3 3 4 4 5]
```

Matrices in Python:

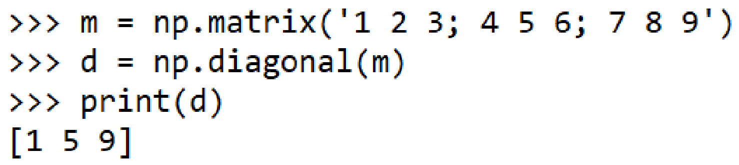
```
>>> import numpy as np
>>> a = np.matrix('1 2; 3 4')
>>> print("Via string input : \n", a, "\n\n")
Via string input :
 [[1 2]
 [3 4]]
```



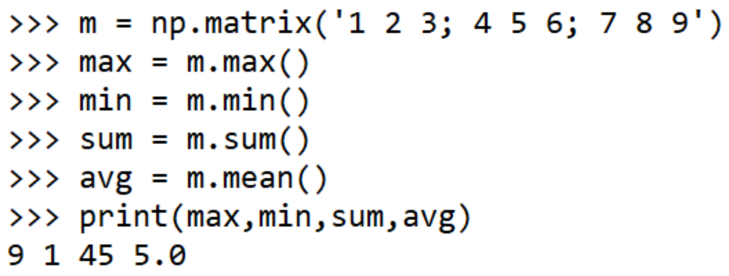
```
>>> b = np.matrix([[5, 6, 7], [3, 4, 6]])
>>> print("Via array-like input : \n", b)
Via array-like input :
 [[5 6 7]
 [3 4 6]]
```



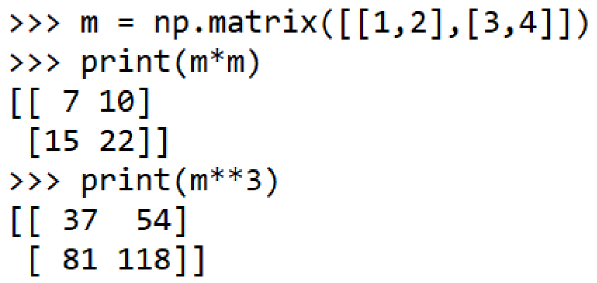
```
>>> l = [[1, 2, 3], [3, 6, 9], [2, 4, 6]]
>>> a = np.matrix(l)
>>> print(a)
[[1 2 3]
 [3 6 9]
 [2 4 6]]
```



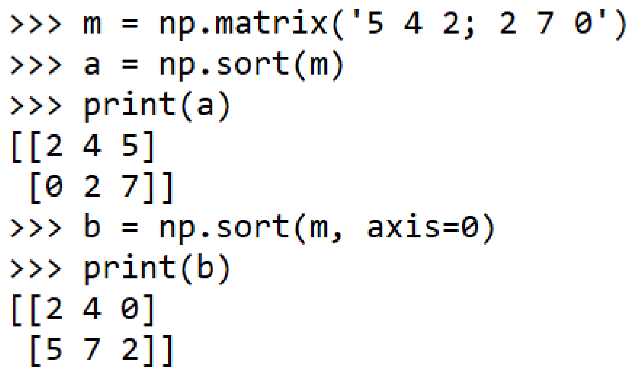
```
>>> m = np.matrix('1 2 3; 4 5 6; 7 8 9')
>>> d = np.diagonal(m)
>>> print(d)
[1 5 9]
```



```
>>> m = np.matrix('1 2 3; 4 5 6; 7 8 9')
>>> max = m.max()
>>> min = m.min()
>>> sum = m.sum()
>>> avg = m.mean()
>>> print(max,min,sum,avg)
9 1 45 5.0
```



```
>>> m = np.matrix([[1,2],[3,4]])
>>> print(m*m)
[[ 7 10]
 [15 22]]
>>> print(m**3)
[[ 37  54]
 [ 81 118]]
```



```
>>> m = np.matrix('5 4 2; 2 7 0')
>>> a = np.sort(m)
>>> print(a)
[[2 4 5]
 [0 2 7]]
>>> b = np.sort(m, axis=0)
>>> print(b)
[[2 4 0]
 [5 7 2]]
```

```
>>> m = np.matrix('1 2 3; 4 5 6; 7 8 9')
>>> print(m)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> t = m.transpose()
>>> print(t)
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

```
>>> a = np.matrix('1 2 3; 4 5 6')
>>> b = np.matrix('2 2 2; 1 -2 2')
>>> c=a+b
>>> print(c)
[[3 4 5]
 [5 3 8]]
>>> d=a/b
>>> print(d)
[[ 0.5  1.   1.5]
 [ 4.  -2.5  3. ]]
```

```
>>> a = np.matrix('1 2 3; 4 5 6')
>>> b = np.matrix('1 0; 0 2; 1 -1')
>>> c=a*b
>>> print(c)
[[ 4  1]
 [10  4]]
```

<u>**Conclusion**</u>:

Thus, we have understood the concepts of basic arrays, two-dimensional arrays and multidimensional arrays using the array of NumPy. We also learnt methods of array and matrix creation and various operations on them, along with numerous built-in functions in array and understood the hands-on practical use of them.