Numpy

- NumPy is a Python package.
- NumPy stands for 'Numerical Python'.
- It is core library for scientific computing.
- It provides a high-performance multidimensional array object, and tools for working with these arrays
- Basic operations using NumPy:
  - Mathematical and logical operations on arrays.
  - Operations related to linear algebra.
  - Random number generation.
  - Fourier transforms and shape manipulation etc.

Array

- The central feature of NumPy is the ndarray object class.
- Arrays are similar to lists in Python, except that every element of an array must be of the same type, typically a numeric type.
- With large amounts of numeric data, arrays make operations very fast and efficient as compare to lists.

```
>>>import numpy as np
>>>arr = np.array([1,2,3,4,5], float)
>>>arr
array([1. 2. 3. 4. 5.])
>>>type(arr)
<class 'numpy.ndarray'>
```

- In np.array() the second argument is optional represents the desired data-type for the array. If not given, then the data type will be determined as the minimum data type required to hold the objects in the sequence.

Importing the NumPy library

```
>>>import numpy
```

- For large amounts of calls to NumPy functions, it can become tedious to write numpy.Y over and over again.
- A common practice is to import numpy under the brief name np

```
>>>import numpy as np
```

Array

- Array elements are accessed, sliced, and manipulated just like lists:

```
>>>import numpy as np
>>>arr = np.array([1,2,3,4,5])
>>>print(arr[:3])
[1 2 3]
>>>arr[3]
4
>>>arr[-1]
5
>>>arr[1] = 2.5
>>>print(arr)
[1 2 3 4 5]
>>arr[1] = 0.5
>>>print(arr)
[1 0 3 4 5]
```

Array

- A two-dimensional array (e.g., a matrix) can be created using numpy as follow:

```
>>>import numpy as np
>>>arr = np.array([[1,2,3,4,5],[6,7,8,9,10]])
>>>arr[0]
array([1, 2, 3, 4, 5])
>>>arr[0][0]
1
>>>arr[1][2]
8
>>>arr[-1][-1]
10
>>arr[:,2]
array([3, 8])
```

Array

```
# Create a 2x2 identity matrix
>>>arr4 = np.eye(2)
>>>print(arr4)
[[1. 0.]
 [0. 1.]]
# Create an array filled with random values
>>arr5 = np.random.random((1,2))
>>>print(arr5)
[[0.00739397, 0.35334824]]
```

Array

- Numpy methods to create arrays:

```
>>>import numpy as np
# Create an array of all zeros
>>>arr1 = np.zeros((2,2))
>>>print(arr1)
[[0. 0.]
 [0. 0.]]
# Create an array of all ones
>>>arr2 = np.ones((3,2))
>>>print(arr2)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
# Create a constant array
>>>arr3 = np.full((2,2), 5)
>>>print(arr3)
[[5. 5.]
 [5. 5.]]
```

Numpy array methods

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]],
    float)
# shape property returns the dimension of array
>>>arr1.shape
(2, 5)
# dtype tells the type of the values stored by an array
>>>print(arr1.dtype)
float64
# reshape method returns array of new specified dimension
>>>arr2 = arr1.reshape((5,2))
>>>print(arr2)
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9. 10.]]
```

Numpy array methods

```
# transpose method returns transpose versions of arrays
>>>print(arr1.arr1.transpose())
[[ 1.,  6.]
 [ 2.,  7.]
 [ 3.,  8.]
 [ 4.,  9.]
 [ 5., 10.]]
# flatten method returns One-dimensional versions of
    multi-dimensional array
>>>print(arr1.flatten())
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

Numpy concatenate() function

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2], [3, 4]], float)
>>>arr2 = np.array([[5, 6], [7,8]], float)
>>>print(np.concatenate((arr1,arr2), axis = 0))
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
>>>arr3 = np.concatenate((arr1,arr2),axis = 1)
>>>arrr3
array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])
```

Numpy concatenate() function

- concatenate() function is used to join two or more arrays of the same shape along a specified axis.

Syntax:

numpy.concatenate((a1,a2,...), axis)

(a1,a2,...) : sequence of arrays of the same type

axis (optional): Axis along which arrays have to be joined. Default is 0.

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2], [3, 4]], float)
>>>arr2 = np.array([[5, 6], [7,8]], float)
>>>print(np.concatenate((arr1,arr2)))
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
```

Array mathematics

- When standard mathematical operations are used with arrays, they are applied on element-by-element basis.

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2], [3, 4]])
>>>arr2 = np.array([[5, 6], [7,8]])
>>>print(arr1/arr2)
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
>>>print(arr1 + 5)
[[6 7]
 [8 9]]
>>>arr3 = np.array([5,6])
>>>print(arr1 * arr3)
[[ 5 12]
 [15 24]]
>>>arr4 = np.array([[5,6],[7,8], [9, 10]])
>>>print(arr1 * arr4)
ValueError: operands could not be broadcast together with
    shapes (2,2) (3,2)
```

Array iteration

- Iterate over arrays is possible in a manner similar to that of lists:

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
>>>for x in arr:
...    print(x)
[1. 2.]
[3. 4.]
[5. 6.]
[7. 8.]
>>>for (x, y) in arr:
...    print(x + y)
3.0
7.0
11.0
15.0
```

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
#product operation
>>>print(arr.prod())
40320.0
>>>print(arr.prod(axis = 0))
[105. 384.]
>>>print(arr.prod(axis = 1))
[ 2. 12. 30. 56.]
>>>print(np.prod(arr))
40320.0
>>>print(np.prod(arr, axis = 1))
[ 2. 12. 30. 56.]
>>>print(np.prod(arr, axis = 0))
[105. 384.]
```

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
#sum operation
>>>print(arr.sum())
36.0
>>>print(arr.sum(axis = 0))
[16. 20.]
>>>print(arr.sum(axis = 1))
[ 3.  7. 11. 15.]
>>>print(np.sum(arr))
36.0
>>>np.sum(arr, axis = 1)
[ 3.  7. 11. 15.]
>>>np.sum(arr, axis = 0)
[16. 20.]
```

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
# mean
>>>print(arr.mean())
4.5
>>>print(arr.mean(axis = 0))
[4. 5.]
>>>print(arr.mean(axis = 1))
[1.5 3.5 5.5 7.5]
>>>print(np.mean(arr))
4.5
>>>print(np.mean(arr, axis = 1))
[1.5 3.5 5.5 7.5]
>>>print(np.mean(arr, axis = 0))
[4. 5.]
```

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
# variance
>>>print(arr.var())
5.25
>>>print(arr.var(axis = 0))
[5. 5.]
>>>print(arr.var(axis = 1))
[0.25 0.25 0.25 0.25]
# standard deviation
>>>print(arr.std())
2.29128784747792
>>>print(arr.std(axis = 0))
[2.23606798 2.23606798]
>>>print(arr.std(axis = 1))
[0.5 0.5 0.5 0.5]
```

- We can also find variance and standard deviation using np.var() and np.std().

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[5, 7, 4], [8, 2, 1]], float)
# argmin
>>>print(arr.argmin())
5
>>>print(arr.argmin(axis = 0))
[0 1 1]
>>>print(arr.argmin(axis = 1))
[2 2]
# argmax
>>>print(arr.argmax())
3
>>>print(arr.argmax(axis = 0))
[1 0 0]
>>>print(arr.argmax(axis = 1))
[1 0]
```

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
# minimum
>>>print(arr.min())
1.0
>>>print(arr.min(axis = 0))
[1. 2.]
>>>print(arr.min(axis = 1))
[1. 3. 5. 7.]
# maximum
>>>print(arr.max())
8.0
>>>print(arr.max(axis = 0))
[7. 8.]
>>>print(arr.max(axis = 1))
[2. 4. 6. 8.]
```

- We can also use np.min() and np.max() to find minimum and maximum.

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[5, 7, 4], [8, 2, 1]], float)
>>>arr1 = arr.copy()
>>>arr2 = arr.copy()
# sort
>>>arr.sort()
>>>print(arr)
[[4. 5. 7.]
 [1. 2. 8.]]
>>>arr1.sort(axis = 0)
print(arr1)
[[5. 2. 1.]
 [8. 7. 4.]]
>>>arr2.sort(axis = 1)
>>>print(arr2)
[[4. 5. 7.]
 [1. 2. 8.]]
```

Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[5, 7, 4], [8, 2, 1], [3, 9, 6]], float)
# diagonal
>>>print(arr.diagonal())
[5. 2. 6.]
>>>print(np.diag(arr))
[5. 2. 6.]
>>>arr = np.array([[5, 7, 4], [8, 2, 1]], float)
>>>print(arr.diagonal())
[5. 2.]
```

Vector and matrix mathematics

```
>>>import numpy as np
>>>a = np.array([[4, 1], [3, 2]], float)
>>>b = np.array([2, 1], float)
>>>c = np.array([[1, 1], [4, 1]], float)
# dot product
>>>print(np.dot(b, a))
[11.  4.]
>>>print(np.dot(a, b))
[9. 8.]
>>>print(np.dot(a, c))
[[ 8.  5.]
 [11.  5.]]
>>>print(np.dot(c, a))
[[ 7.  3.]
 [19.  6.]]
```

Comparison operators

```
>>>import numpy as np
>>>arr1 = np.array([5, 7, 4])
>>>arr2 = np.array([2, 8, 3])
# comparision
>>>print(arr1 > arr2)
[ True False  True]
>>>print(arr1 < arr2)
[False  True False]
>>>print(arr1 != arr2)
[ True  True  True]
>>>print(arr1 > 4)
[ True  True False]
```

Outer product, inner product, cross product

```
>>>import numpy as np
>>>a = np.array([1, 2, 0], float)
>>>b = np.array([1, 2, 1], float)
# outer product
>>>print(np.outer(b, a))
[[1. 2. 0.]
 [2. 4. 0.]
 [1. 2. 0.]]
>>>print(np.outer(a, b))
[[1. 2. 1.]
 [2. 4. 2.]
 [0. 0. 0.]]
# inner product
>>>print(np.inner(a, b))
5.0
>>>print(np.inner(b, a))
5.0
#cross product
>>>print(np.cross(a, b))
[ 2. -1.  0.]
```

numpy.linalg

```
>>>import numpy as np
>>>a = np.array([[1, 2, 1], [1, 3, 1], [1, 2, 0]], float)
# determinant
>>>print(np.linalg.det(a))
-1.0
# inverse matrix
>>>b = np.linalg.inv(a)
>>>print(b)
[[ 2. -2.  1.]
 [-1.  1.  0.]
 [ 1. -0. -1.]]
# dot product
>>>print(np.dot(a, b))
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

numpy.linalg

```
>>>import numpy as np
>>>a = np.array([[-2, -4, 2], [-2, 1, 2], [4, 2, 5]],
    float)
# eigen value and eigen vectors
>>>vals, vecs = np.linalg.eig(a)
>>>print(vals)
[-5.  3.  6.]
# normalized eigen vectors as columns of vecs
>>>print(vecs)
[[ 0.81649658  0.53452248  0.05842062]
 [ 0.40824829 -0.80178373  0.35052374]
 [-0.40824829 -0.26726124  0.93472998]]
>>>print(np.dot(a,vecs[:,0]))
[-4.0824829  -2.04124145  2.04124145]
>>>print(vals[0]*vecs[:,0])
[-4.0824829  -2.04124145  2.04124145]
```