

**2****Python Numpy****Topics Covered**

- |                                      |                                  |
|--------------------------------------|----------------------------------|
| 1. NumPy Introduction                | 9. NumPy Array Iteration         |
| 2. Environment Setup                 | 10. NumPy Bitwise Operators      |
| 3. NumPy Narray                      | 11. NumPy String Functions       |
| 4. NumPy Data Types                  | 12. NumPy Mathematical Functions |
| 5. NumPy Array Creation              | 13. Statistical Functions        |
| 6. Array from Existing Data          | 14. Sorting & Searching          |
| 7. Arrays within the numerical range | 15. Copies and Views             |
| 8. NumPy Broadcasting                | 16. Matrix Library               |
|                                      | 17. NumPy Linear Algebra         |

**NumPy Introduction:**

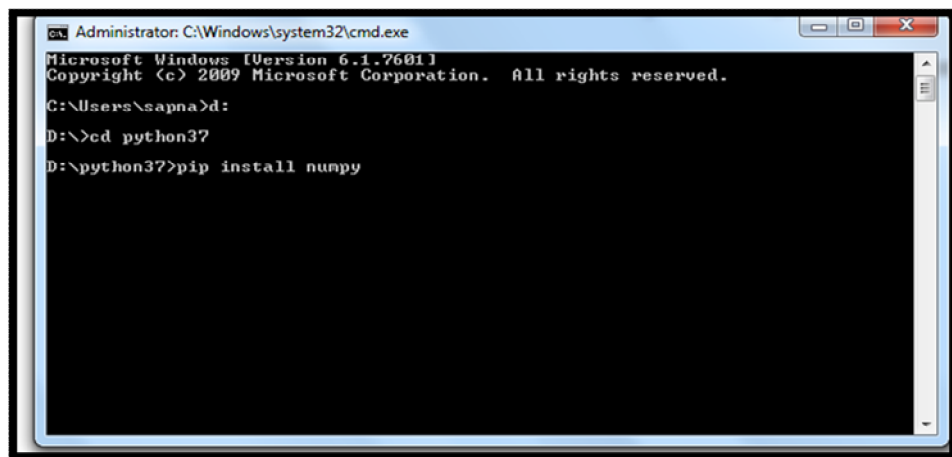
- NumPy is a Python package. It stands for 'Numerical Python'.
- It is a library consisting of multidimensional array objects and a collection of routines for processing of array.
- NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.
- NumPy is a general-purpose array-processing package.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.
- It is the fundamental package for scientific computing with Python.
- Besides its obvious scientific uses, Numpy can also be used as an efficient multi-dimensional container of generic data.
- Numeric, the ancestor of NumPy, was developed by Jim Hugunin.
- Another package Numarray was also developed, having some additional functionality.
- 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.
- NumPy is often used along with packages like SciPy (Scientific Python) and Matplotlib (plotting library).
- This combination is widely used as a replacement for MatLab, a popular platform for technical computing.

**Operations using NumPy:**

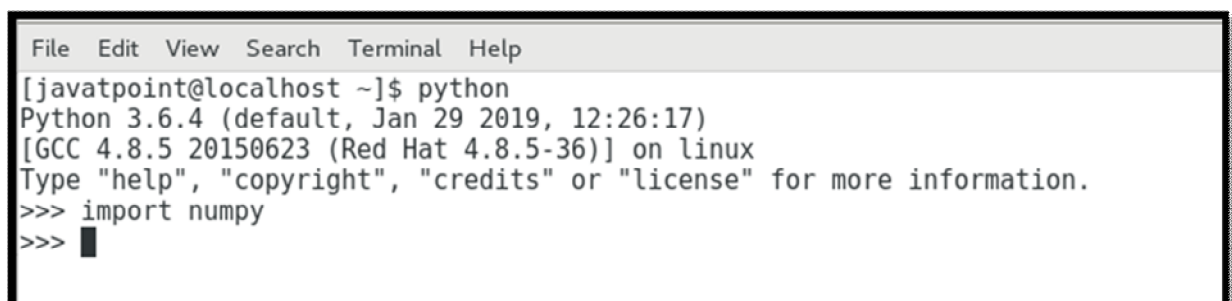
- Using NumPy, a developer can perform the following operations –
- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

**NumPy Environment:**

- Standard Python distribution doesn't come bundled with NumPy module.
- A lightweight alternative is to install NumPy using popular Python package installer, pip.
- If you have Python and PIP already installed on a system, then installation of NumPy is very easy.
- Install it using this command: **C:\Users\Your Name>pip install numpy**
- If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, and Spyder etc.
- On the Windows operating system, The SciPy stack is provided by the Anaconda which is a free distribution of the Python SciPy package.
- It can be downloaded from the official website: <https://www.anaconda.com/>.
- It is also available for Linux and Mac.
- 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).

**Import NumPy**

- Once NumPy is installed, import it in your applications by adding the import keyword: **import numpy**
- Now NumPy is imported and ready to use.



**Example:****Example:**

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

**Output:****NumPy as np:**

- NumPy is usually imported under the np alias.
- Alias: In Python alias are alternate names for referring to the same thing.
- Create an alias with the as keyword while importing:  
**import numpy as np**
- Now the NumPy package can be referred to as np instead of numpy.

**Example:**

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

**Output:****NumPy Ndarray:**

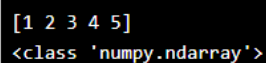
- The most important object defined in NumPy is an N-dimensional array type called ndarray.
- It describes the collection of items of the same type.
- Items in the collection can be accessed using a zero-based index.
- Every item in an ndarray takes the same size of block in the memory.
- Each element in ndarray is an object of data-type object (called dtype).
- The basic ndarray is created using an array function in NumPy as follows –numpy.array
- Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.
- We can create a NumPy Ndarray object by using the array () function.  
**numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)**
- The above constructor takes the following parameters :

Sr.No.	Parameter & Description
1	<b>Object</b> Any object exposing the array interface method returns an array, or any (nested) sequence.
2	<b>Dtype</b> Desired data type of array, optional
3	<b>Copy</b> Optional. By default (true), the object is copied
4	<b>Order</b> C (row major) or F (column major) or A (any) (default)
5	<b>Subok</b> By default, returned array forced to be a base class array. If true, sub-classes passed through
6	<b>Ndmin</b> Specifies minimum dimensions of resultant array

- Take a look at the following examples to understand better.

**Example:**

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

**Output:**

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

- **Type ():** This built-in Python function tells us the type of the object passed to it.
- Like in above code it shows that arr is numpy.ndarray type.
- To create an Nddarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an Nddarray:

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

**NumPy Data Types:**

- By default Python have these data types:
- **Strings** - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- **Integer** - used to represent integer numbers. e.g. -1, -2, -3
- **Float** - used to represent real numbers. e.g. 1.2, 42.42

- **Boolean** - used to represent True or False.
- **Complex** - used to represent complex numbers. e.g.  $1.0 + 2.0j$ ,  $1.5 + 2.5j$

### Data Types in NumPy:

- NumPy has some extra data types, and refer to data types with one character, like **i** for integers, **u** for unsigned integers etc.
- Below is a list of all data types in NumPy and the characters used to represent them.
  1. **i** - integer
  2. **b** - Boolean
  3. **u** - unsigned integer
  4. **f** - float
  5. **c** - complex float
  6. **m** - time delta
  7. **M** - date time
  8. **O** - object
  9. **S** - string
  10. **U** - Unicode string
  11. **V** - fixed chunk of memory for other type ( void )

### Checking the Data Type of an Array:

- The NumPy array object has a property called `dtype` that returns the data type of the array:

### Example:- Get the data type of an array object:

1. 

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

### Output:



2. 

```
import numpy as np
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

### Output:



**Converting Data Type on Existing Arrays:**

- The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.
- The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.
- The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

**Example: Change data type from float to integer by using 'i' as parameter value**

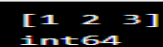
```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

**Output:**

```
[1 2 3]
int32
```

**Example: Change data type from float to integer by using int as parameter value**

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype(int)
print(newarr)
print(newarr.dtype)
```

**Output:**

```
[1 2 3]
int64
```

**NumPy Array Creation:**

- NumPy is used to work with arrays. The array object in NumPy is called ndarray.
- We can create a NumPy ndarray object by using the `array ()` function.
- Dimensions in Arrays:-A dimension in arrays is one level of array depth (nested arrays).

**Example: Create a 0-D array with value 42**

```
import numpy as np
arr = np.array(42)
print(arr)
```

**1-D Arrays:**

- An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

- These are the most common and basic arrays.

**Example :Create a 1-D array containing the values 1,2,3,4,5**

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

**2-D Arrays:**

- An array that has 1-D arrays as its elements is called a 2-D array.
- These are often used to represent matrix or 2nd order tensors.

**Example: Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6**

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

**3-D arrays:**

- An array that has 2-D arrays (matrices) as its elements is called 3-D array.
- These are often used to represent a 3rd order tensor.

**Example: Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:**

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

- The following examples show how to use the built-in range () function to return a list object.

**Example:**

```
# create list object using range function
import numpy as np
list = range(5)
print list
```

**Output:**

```
[0, 1, 2, 3, 4]
```

- Python has a set of built-in methods that you can use on lists/arrays.

**Methods for array creation in Numpy**

Methods	Syntax	Parameter Values	Example
<b>append()</b> Adds an element at the end of the list.	list.append(element)	<b>Element:</b> Required. An element of any type (string, number, object etc.)	fruits = ['apple', 'banana', 'cherry'] fruits.append("orange")
<b>clear()</b> Removes all the	list.clear()	No parameters	fruits = ['apple', 'banana', 'cherry', '']

elements from the list			orange'] fruits.clear()
<b>copy()</b> Returns a copy of the list.	list.copy()	No parameters	fruits = ['apple', 'banana', 'cherry', 'orange'] x = fruits.copy()
<b>count()</b> Returns the number of elements with the specified value	list.count(value)	<b>Value:</b> Required. Any type (string, number, list, tuple, etc.). The value to search for.	points = [1, 4, 2, 9, 7, 8, 9, 3, 1]  x = points.count(9)
<b>index()</b> Returns the index of the first element with the specified value	list.index(element)	<b>Element:</b> Required. Any type (string, number, list, etc.). The element to search for	fruits = ['apple', 'banana', 'cherry']  x = fruits.index("cherry")
<b>extend()</b> Add the elements of a list (or any iterable), to the end of the current list	list.extend(iterable)	<b>Iterable:</b> Required. Any iterable (list, set, tuple, etc.)	fruits = ['apple', 'banana', 'cherry'] cars = ['Ford', 'BMW', 'Volvo'] fruits.extend(cars)
<b>pop()</b> Removes the element at the specified position	list.pop(pos)	<b>Pos:</b> Optional: A number specifying the position of the element you want to remove, default value is -1, which returns the last item	fruits = ['apple', 'banana', 'cherry']  fruits.pop(1)
<b>insert()</b> Adds an element at the specified position	list.insert(pos, element)	<b>Pos:</b> Required. A number specifying in which position to insert the value <b>Element:</b> Required. An element of any type (string, number, object)	fruits = ['apple', 'banana', 'cherry']  fruits.insert(1, "orange")
<b>Remove ()</b> Removes the first item with the specified value	list.remove(element)	<b>Element:</b> Required. Any type (string, number, list etc.) The element you want to remove	fruits = ['apple', 'banana', 'cherry']  fruits.remove("banana")
<b>reverse()</b> Reverses the order	list.reverse()	No parameters	fruits = ['apple', 'banana', 'cherry']



of the list			fruits.reverse()
<b>sort()</b> Sorts the list	list.sort(reverse = True False, key=myFunc)	<b>Reverse:</b> Optional.Reverse=True will sort the list descending. Default is reverse=False <b>Key:</b> Optional. A function to specify the sorting criteria(s)	cars = ['Ford', 'BMW', 'Volvo'] cars.sort()
<b>empty()</b> Return a new array of given shape and type, without initializing entries	numpy.empty (shape, dtype = float, order = 'C')	<b>shape:</b> Number of rows <b>order:</b> C_contiguous or F_contiguous <b>dtype:</b> [optional, float (by Default)] Data type of returned array.	import numpy as geek a = geek.empty([2, 2], dtype = int) print("\matrix a : \n", a)
<b>zeros()</b> Return a new array of given shape and type, filled with zeros	numpy.zeros (shape, dtype = None, order = 'C')	<b>shape:</b> integer or sequence of integers <b>order:</b> C_contiguous or F_contiguous <b>dtype:</b> [optional, float(byDeafult)]	Import numpy as np arr=np.zeros(5) or arr=np.zeros((3,4)) arr
<b>ones()</b> Return a new array of given shape and type, filled with ones.	numpy.ones (shape, dtype = None , order = 'C')	<b>shape:</b> integer or sequence of integers <b>order:</b> C_contiguous or F_contiguous <b>dtype:</b> [optional, float(byDeafult)]	Import numpy as np arr=np.ones(5) or arr=np.ones((5,4)) arr
<b>eye()</b> Return a 2-D array with ones on the diagonal and zeros elsewhere.	numpy.eye (R, C = None, k = 0, dtype = type <'float'>)	<b>R</b> : Number of rows <b>C</b> : [optional] Number of columns; By default M = N <b>k</b> : [int, optional, 0 by default]	import numpy as np arr=np.eye(3) or arr=np.eye((3,4)) arr
<b>diag()</b> Used to creates a two dimensional array with all the diagonal elements	diag(number_of _values)	No parameters	import numpy as np Arr=np.diag([1,5,3,7]) Arr

as the given value and rest as 0			
<b>randint()</b> Used to generate a random number between a given range.	rand(min,max, total_values)	No parameters	import numpy as np arr=np.random.randint(1,10,3) Arr
<b>rand ()</b> Used to generate a random value between 0 to 1.	rand(number_of _values)	No parameters	import numpy as np arr=np.random.rand(5) or arr=np.random.rand (2,3) arr
<b>randn()</b> Used to generate a random values close to 0(zero).	rand(number_of _values)	No parameters	import numpy as np arr= np.random.randn(5) arr

**Numpy array from existing data:**

- NumPy provides us the way to create an array by using the existing data.

**1. numpy.asarray**

- This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object.

**Syntax:**

numpy.asarray(sequence, dtype = None, order = None)

- It accepts the following parameters.

- sequence:** It is the python sequence which is to be converted into the python array.
- dtype:** It is the data type of each item of the array.
- order:** It can be set to C or F. The default is C.

**Example: creating numpy array using the list**

```
import numpy as np
l=[1,2,3,4,5,6,7]
a = np.asarray(l);
print(type(a))
print(a)
```

**Output:**

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

**Example: creating a numpy array using Tuple**

```
import numpy as np
l=(1,2,3,4,5,6,7)
a = np.asarray(l);
Print (type (a))
print (a)
```

**Output:**

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

**Example: creating a numpy array using more than one list**

```
import numpy as np
l=[[1,2,3,4,5,6,7],[8,9]]
a = np.asarray(l);
print (type(a))
print(a)
```

**Output:**

```
<class 'numpy.ndarray'>
[list([1, 2, 3, 4, 5, 6, 7]) list([8, 9])]
```

**2. numpy.frombuffer**

- This function is used to create an array by using the specified buffer.

**Syntax:**

```
numpy.frombuffer (buffer, dtype = float, count = -1, offset = 0)
```

- It accepts the following parameters.
  1. **buffer:** It represents an object that exposes a buffer interface.
  2. **dtype:** It represents the data type of the returned data type array. The default value is 0.
  3. **count:** It represents the length of the returned ndarray. The default value is -1.
  4. **offset:** It represents the starting position to read from. The default value is 0.

**Example:**

```
import numpy as np
l = b'hello world'
print(type(l))
a = np.frombuffer(l, dtype = "S1")
print(a)
print(type(a))
```

**Output:**

```
<class 'bytes'>
[b'h' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd']
<class 'numpy.ndarray'>
```

**3. numpy.fromiter**

- This routine is used to create a ndarray by using an iterable object.
- It returns a one-dimensional ndarray object.

**Syntax:**

```
numpy.fromiter(iterable, dtype, count = - 1)
```

- It accepts the following parameters.

1. **Iterable:** It represents an iterable object.
2. **dtype:** It represents the data type of the resultant array items.
3. **count:** It represents the number of items to read from the buffer in the array.

**Example**

```
import numpy as np
list = [0,2,4,6]
it = iter(list)
x = np.fromiter(it, dtype = float)
print(x)
print(type(x))
```

**Output:**

```
[0. 2. 4. 6.]
<class 'numpy.ndarray'>
```

**Numpy Arrays within the numerical range:**

- This section illustrates how the numpy arrays can be created using some given specified range.

**1. Numpy.arange:**

- It creates an array by using the evenly spaced values over the given interval.

**Syntax:**

```
numpy.arange(start, stop, step, dtype)
```

- It accepts the following parameters.
  1. **start:** The starting of an interval. The default is 0.
  2. **stop:** represents the value at which the interval ends excluding this value.
  3. **step:** The number by which the interval values change.
  4. **dtype:** the data type of the numpy array items.

**Example:**

```
import numpy as np
arr = np.arange(0,10,2,float)
print(arr)
```

**Output:**

```
[0. 2. 4. 6. 8.]
```

**Example**

```
import numpy as np
arr = np.arange(10,100,5,int)
print("The array over the given range is ",arr)
```

**Output:**

```
The array over the given range is [10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]
```

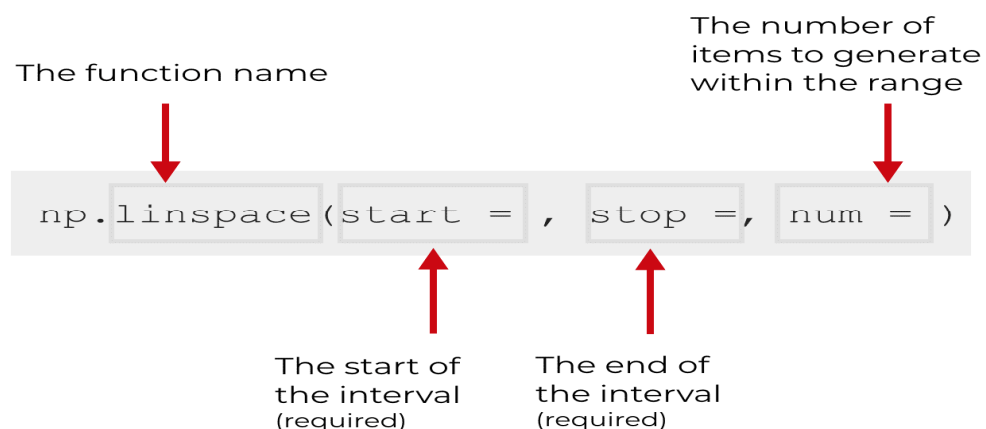
**2. NumPy.linspace:**

- It is similar to the `arrange` function. However, it doesn't allow us to specify the step size in the syntax.
- Instead of that, it only returns evenly separated values over a specified period. The system implicitly calculates the step size.

**Syntax:**

`numpy.linspace(start, stop, num, endpoint, retstep, dtype)`

- It accepts the following parameters.
- 1. **start:** It represents the starting value of the interval.
- 2. **stop:** It represents the stopping value of the interval.
- 3. **num:** The amount of evenly spaced samples over the interval to be generated. The default is 50.
- 4. **endpoint:** Its true value indicates that the stopping value is included in the interval.
- 5. **retstep:** This has to be a boolean value. Represents the steps and samples between the consecutive numbers.
- 6. **dtype:** It represents the data type of the array items.

**Example:**

```
import numpy as np
arr = np.linspace(10, 20, 5)
print("The array over the given range is ",arr)
```

**Output:**

The array over the given range is [10. 12.5 15. 17.5 20.]

**Example:**

```
import numpy as np
arr = np.linspace(10, 20, 5, endpoint = False)
print("The array over the given range is ",arr)
```

**Output:**

The array over the given range is [10. 12. 14. 16. 18.]

**3. numpy.logspace:**

- It creates an array by using the numbers that are evenly separated on a log scale.

**Syntax:**

`numpy.logspace(start, stop, num, endpoint, base, dtype)`

- It accepts the following parameters.
1. **start:** It represents the starting value of the interval in the base.
  2. **stop:** It represents the stopping value of the interval in the base.
  3. **num:** The number of values between the range.
  4. **endpoint:** It is a boolean type value. It makes the value represented by stop as the last value of the interval.
  5. **base:** It represents the base of the log space.
  6. **dtype:** It represents the data type of the array items.

**Example:**

```
import numpy as np
arr = np.logspace(10, 20, num = 5, endpoint = True)
print("The array over the given range is ",arr)
```

**Output:**

The array over the given range is [1.00000000e+10 3.16227766e+12 1.00000000e+15 3.16227766e+17 1.00000000e+20]

**Example:**

```
import numpy as np
arr = np.logspace(10, 20, num = 5, base = 2, endpoint = True)
print("The array over the given range is ",arr)
```

**Output:**

The array over the given range is [1.02400000e+03 5.79261875e+03 3.27680000e+04 1.85363800e+05 1.04857600e+06]

**NumPy Broadcasting:**

- In Mathematical operations, we may need to consider the arrays of different shapes.
- NumPy can perform such operations where the array of different shapes is involved.
- For example, if we consider the matrix multiplication operation, if the shape of the two matrices is the same then this operation will be easily performed.
- However, we may also need to operate if the shape is not similar.

**Broadcasting Rules:**

- Broadcasting is possible if the following cases are satisfied.
1. The smaller dimension array can be appended with '1' in its shape.
  2. Size of each output dimension is the maximum of the input sizes in the dimension.
  3. An input can be used in the calculation if its size in a particular dimension matches the output size or its value is exactly 1.

4. If the input size is 1, then the first data entry is used for the calculation along the dimension.
- Broadcasting can be applied to the arrays if the following rules are satisfied.
    1. All the input arrays have the same shape.
    2. Arrays have the same number of dimensions, and the length of each dimension is either a common length or 1.
    3. Array with the fewer dimension can be appended with '1' in its shape.

**Example: to multiply two arrays**

```
import numpy as np
a = np.array([1,2,3,4,5,6,7])
b = np.array([2,4,6,8,10,12,14])
c = a*b;
print(c)
```

**Output:**

```
[ 2  8 18 32 50 72 98]
```

**Example:**

```
import numpy as np
a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
b = np.array([2,4,6,8])
print("\nprinting array a..")
print(a)
print("\nprinting array b..")
print(b)
print("\nAdding arrays a and b ..")
c = a + b;
print(c)
```

**Output:**

```
printing array a..
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]
```

```
Printing array b..
[2 4 6 8]
```

```
Adding arrays a and b...
[[ 3  6  9 12]
 [ 4  8 11 14]
 [12 24 45 11]]
```

**NumPy Array Iteration:**

- NumPy provides an iterator object, i.e., `nditer` which can be used to iterate over the given array using python standard Iterator interface.
- NumPy package contains an iterator object `numpy.nditer`.
- It is an efficient multidimensional iterator object using which it is possible to iterate over an array.
- Each element of an array is visited using Python's standard Iterator interface.

**Example:**

```
import numpy as np
a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
print("Printing array:")
print(a);
```



```
print("Iterating over the array:")
for x in np.nditer(a):
    print(x,end=' ')
```

**Output:**

```
Printing array:
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]
Iterating over the array:
1 2 3 4 2 4 5 6 10 20 39 3
```

- Let's iterate over the transpose of the array given in the above example.

**Example:**

```
import numpy as np
a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
print("Printing the array:")
print(a)
print("Printing the transpose of the array:")
at = a.T
print(at)
#this will be same as previous
for x in np.nditer(at):
    print(print("Iterating over the array:"))
for x in np.nditer(a):
    print(x,end=' ')
```

**Output:**

```
Printing the array:
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]
Printing the transpose of the array:
[[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]
1 2 3 4 2 4 5 6 10 20 39 3
```

**Order of Iteration:**

- As we know, there are two ways of storing values into the numpy arrays:
  1. F-style order
  2. C-style order
- Let's see an example of how the numpy Iterator treats the specific orders (F or C).

**Example:**

```
import numpy as np
```

```
a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
print("\nPrinting the array:\n")
print(a)
print("\nPrinting the transpose of the array:\n")
at = a.T
print(at)
print("\nIterating over the transposed array\n")
for x in np.nditer(at):
    print(x, end= ' ')
print("\nSorting the transposed array in C-style:\n")
c = at.copy(order = 'C')
print(c)
print("\nIterating over the C-style array:\n")
for x in np.nditer(c):
    print(x,end=' ')
d = at.copy(order = 'F')
print(d)
print("Iterating over the F-style array:\n")
for x in np.nditer(d):
    print(x,end=' ')
```

**Output:**

Printing the array:

```
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]
```

Printing the transpose of the array:

```
[[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]
```

Iterating over the transposed array

```
1 2 3 4 2 4 5 6 10 20 39 3
```

Sorting the transposed array in C-style:

```
[[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]
```

Iterating over the C-style array:

```
1 2 10 2 4 20 3 5 39 4 6 3
```

```
[[ 1 2 10]
 [ 2 4 20]
 [ 3 5 39]
 [ 4 6 3]]
```

Iterating over the F-style array:

```
1 2 3 4 2 4 5 6 10 20 39 3
```

### **NumPy Bitwise Operators:**

- Bitwise methods in numpy perform operations on bits.
- The main reason for using bitwise method is to combine values to form a new value.
- Numpy provides the following bitwise operators.

SN	Operator	Description
1	bitwise_and	It is used to calculate the bitwise and operation between the corresponding array elements.
2	bitwise_or	It is used to calculate the bitwise or operation between the corresponding array elements.
3	Invert	It is used to calculate the bitwise not the operation of the array elements.
4	left_shift	It is used to shift the bits of the binary representation of the elements to the left.
5	right_shift	It is used to shift the bits of the binary representation of the elements to the right.

### **1. Bitwise\_and Operation:**

- The NumPy provides the bitwise\_and () function which is used to calculate the bitwise\_and operation of the two operands.
- The bitwise and operation is performed on the corresponding bits of the binary representation of the operands.
- If both the corresponding bit in the operands is set to 1, then only the resultant bit in the AND result will be set to 1 otherwise it will be set to 0.

### **Example:**

```
import numpy as np
a = 10
b = 12
print ("binary representation of a:",bin(a))
print ("binary representation of b:",bin(b))
print("Bitwise-and of a and b: ",np.bitwise_and(a,b))
```

### **Output:**

binary representation of a: 0b1010

binary representation of b: 0b1100

Bitwise-and of a and b: 8

**AND Truth Table:**

- The output of the AND result of the two bits is 1 if and only if both the bits are 1 otherwise it will be 0.

A	B	AND (A, B)
0	0	0
0	1	0
1	0	0
1	1	1

**2. Bitwise\_or Operator:**

- The NumPy provides the bitwise\_or () function which is used to calculate the bitwise or operation of the two operands.
- The bitwise or operation is performed on the corresponding bits of the binary representation of the operands.
- If one of the corresponding bit in the operands is set to 1 then the resultant bit in the OR result will be set to 1; otherwise it will be set to 0.

**Example:**

```
import numpy as np
```

```
a = 50
```

```
b = 90
```

```
print ("binary representation of a:",bin(a))
```

```
print ("binary representation of b:",bin(b))
```

```
print ("Bitwise-or of a and b: ",np.bitwise_or(a,b))
```

**Output:**

```
binary representation of a: 0b110010
```

```
binary representation of b: 0b1011010
```

```
Bitwise-or of a and b: 122
```

**Or Truth Table:**

- The output of the OR result of the two bits is 1 if one of the bits are 1 otherwise it will be 0.

A	B	Or (A, B)
0	0	0
0	1	1

1	0	1
1	1	1

**3. Invert operation:**

- It is used to calculate the bitwise not the operation of the given operand.
- The 2's complement is returned if the signed integer is passed in the function.

**Example:**

```
import numpy as np
arr = np.array([20],dtype = np.uint8)
print("Binary representation:",np.binary_repr(20,8))
print(np.invert(arr))
print("Binary representation: ", np.binary_repr(235,8))
```

**Output:**

```
Binary representation: 00010100
[235]
Binary representation: 11101011
```

**4. Left Shift Operation:**

- It shifts the bits in the binary representation of the operand to the left by the specified position.
- An equal number of 0s are appended from the right.

**Example:**

```
import numpy as np
print("left shift of 20 by 3 bits",np.left_shift(20, 3))
print("Binary representation of 20 in 8 bits",np.binary_repr(20, 8))
print("Binary representation of 160 in 8 bits",np.binary_repr(160,8))
```

**Output:**

```
left shift of 20 by 3 bits 160
Binary representation of 20 in 8 bits 00010100
Binary representation of 160 in 8 bits 10100000
```

**5. Right Shift Operation:**

- It shifts the bits in the binary representation of the operand to the right by the specified position.
- An equal number of 0s are appended from the left.

**Example:**

```
import numpy as np
print("Right shift of 20 by 3 bits",np.right_shift(20, 3))
```

```
print("Binary representation of 20 in 8 bits",np.binary_repr(20, 8))
print("Binary representation of 160 in 8 bits",np.binary_repr(160,8))
```

**Output:**

Right shift of 20 by 3 bits 2

Binary representation of 20 in 8 bits 00010100

Binary representation of 160 in 8 bits 10100000

**NumPy String Functions:**

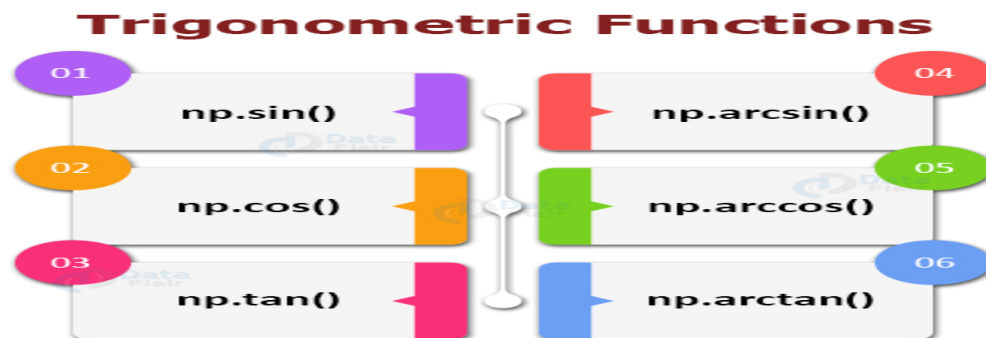
- NumPy contains the following functions for the operations on the arrays of dtype string.

Function	Description	Example
add()	It is used to concatenate the corresponding array elements (strings).	import numpy as np print("Concatenating two string arrays:") print(np.char.add(['welcome','Hi'],[' to P YTHON', ' read python'] ))
Multiply()	It returns the multiple copies of the specified string, i.e., if a string 'hello' is multiplied by 3 then, a string 'hello hello' is returned.	import numpy as np print("Printing a string multiple times:") print(np.char.multiply("hello ",3))
center()	It returns the copy of the string where the original string is centered with the left and right padding filled with the specified number of fill characters.	import numpy as np print("Padding the string through left and right with the fill char *"); #np.char.center(string, width, fillchar) print(np.char.center("Python", 20, '*'))
Capitalize()	It returns a copy of the original string in which the first letter of the original string is converted to the Upper Case.	import numpy as np print("Capitalizing the string ...") print(np.char.capitalize("come mmg"))
title()	It returns the title cased version of the string, i.e., the first letter of each word of the string is converted into the upper case.	import numpy as np print("Converting string into title ...") print(np.char.title("welcome to ty))
lower()	It returns a copy of the string in which all the letters are converted into the lower case.	import numpy as np print("Converting all the characters of the string into lowercase...") print(np.char.lower("WELCOME TO MMG"))
upper()	It returns a copy of the string in which all the letters are converted into the	import numpy as np print("Converting all the characters of

	upper case.	the string into uppercase...") print(np.char.upper("Welcome To mmg "))
split()	It returns a list of words in the string.	import numpy as np print("Splitting the String word by word ..") print(np.char.split("Welcome To TY", sep = " "))
Splitlines()	It returns the list of lines in the string, breaking at line boundaries.	import numpy as np print("Splitting the String line by line..") print(np.char.splitlines("Welcome\nTo House"))
strip()	Returns a copy of the string with the leading and trailing white spaces removed.	import numpy as np str = " welcome to point " print("Original String:",str) print("Removing the leading and trailing whitespaces from the string") print(np.char.strip(str))
join()	It returns a string which is the concatenation of all the strings specified in the given sequence.	import numpy as np print(np.char.join(':', 'HM'))
replace()	It returns a copy of the string by replacing all occurrences of a particular substring with the specified one.	import numpy as np str = "Welcome to Javatpoint" print("Original String:",str) print("Modified String:",end=" ") print(np.char.replace(str, "Welcome to", "www."))
decode()	It is used to decode the specified string element-wise using the specified codec.	import numpy as np enstr = np.char.encode("welcome to jav atpoint", 'cp500') dstr = np.char.decode(enstr, 'cp500') print(enstr) print(dstr)
encode()	It is used to encode the decoded string elemt-wise specified codec..	import numpy as np enstr = np.char.encode("welcome to jav atpoint", 'cp500') dstr = np.char.decode(enstr, 'cp500') print(enstr) print(dstr)

**NumPy Mathematical Functions:**

- Numpy contains a large number of mathematical functions which can be used to perform various mathematical operations.
- The mathematical functions include trigonometric functions, arithmetic functions, and functions for handling complex numbers.

**1. Trigonometric functions:**

- Numpy contains the trigonometric functions which are used to calculate the sine, cosine, and tangent of the different angles in radian.
- The sin, cos, and tan functions return the trigonometric ratio for the specified angles.
- arcsin(), arccos(), and arctan() functions return the trigonometric inverse of the specified angles.
- The numpy.degrees() function can be used to verify the result of these trigonometric functions.

Function	Description	Example
np.sin()	It performs trigonometric sine calculation element-wise.	import numpy as np arr=np.array([0,30,60,90,120,150,180]) print(np.sin(arr*np.pi/180))
np.cos()	It performs trigonometric cosine calculation element-wise.	import numpy as np arr=np.array([0,30,60,90,120,150,180]) print(np.cos(arr*np.pi/180))
np.tan()	It performs trigonometric tangent calculation element-wise.	import numpy as np arr=np.array([0,30,60,90,120,150,180]) print(np.tan(arr*np.pi/180))
np.arcsin()	It performs trigonometric inverse sine element-wise.	import numpy as np arr=np.array([1,2,3]) print(np.arcsin(arr*np.pi/180))
np.arccos()	It performs trigonometric inverse of cosine element-wise.	import numpy as np arr=np.array([1,2,3])



	wise.	<code>print(np.arccos(arr*np.pi/180))</code>
<code>np.arctan()</code>	It performs trigonometric inverse of tangent element-wise.	<code>import numpy as np arr=np.array([1,2,3]) print(np.arctan(arr*np.pi/180))</code>

## 2. Rounding Functions:

### Rounding Functions in NumPy



- The numpy provides various functions that can be used to truncate the value of a decimal float number rounded to a particular precision of decimal numbers.

#### 1. `numpy.around()` function:

- This function returns a decimal value rounded to a desired position of the decimal.

#### Syntax:

`numpy.around(num, decimal)`

- It accepts the following parameters.

Sr. No.	Parameter	Description
1	num	It is the input number.
2	decimal	It is the number of decimals which to the number is to be rounded. The default value is 0. If this value is negative, then the decimal will be moved to the left.

#### Example:

```
import numpy as np
arr = np.array([12.202, 90.23120, 123.020, 23.202])
print("Printing the original array values:",end = " ")
print(arr)
print("Array values rounded off to 2 decimal position",np.around(arr, 2))
print("Array values rounded off to -1 decimal position",np.around(arr, -1))
```

#### Output:

```
Printing the original array values: [12.202  90.2312 123.02  23.202 ]
Array values rounded off to 2 decimal position [12.2  90.23 123.02 23.2 ]
```

Array values rounded off to -2 decimal position [10. 90. 120. 20.]

## 2. **numpy.floor() function:**

- This function is used to return the floor value of the input data which is the largest integer not greater than the input value.

### **Example:**

```
import numpy as np
arr = np.array([12.202, 90.23120, 123.020, 23.202])
print(np.floor(arr))
```

### **Output:**

```
[ 12.  90. 123.  23.]
```

## 3. **numpy.ceil() function:**

- This function is used to return the ceiling value of the array values which is the smallest integer value greater than the array element.

### **Example:**

```
import numpy as np
arr = np.array([12.202, 90.23120, 123.020, 23.202])
print(np.ceil(arr))
```

### **Output:**

```
[ 13.  91. 124.  24.]
```

## **Numpy Statistical functions:**

- Numpy provides various statistical functions which are used to perform some statistical data analysis.
- NumPy has quite a few useful statistical functions for finding minimum, maximum, percentile standard deviation and variance, etc. from the given elements in the array.
- The functions are explained as follows -

## 1. **numpy.amin() and numpy.amax():**

- The numpy.amin() and numpy.amax() functions are used to find the minimum and maximum of the array elements along the specified axis respectively.

### **Example:**

```
import numpy as np
a = np.array([[3,7,5],[8,4,3],[2,4,9]])
print 'Our array is:'
print a
print '\n'
```

```
print 'Applying amin() function:'
print np.amin(a,1)
print '\n'
print 'Applying amin() function again:'
print np.amin(a,0)
print '\n'
print 'Applying amax() function:'
print np.amax(a)
print '\n'
print 'Applying amax() function again:'
print np.amax(a, axis = 0)
```

**Output:**

Our array is:

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

Applying amin() function:

```
[3 3 2]
```

Applying amin() function again:

```
[2 4 3]
```

Applying amax() function:

```
9
```

Applying amax() function again:

```
[8 7 9]
```

**2. numpy.ptp():**

- The name of the function numpy.ptp() is derived from the name peak-to-peak. It is used to return the range of values along an axis.
- numpy.ptp() functions use to plays an important role in statistics by finding out Range of given numbers.
- **Range = max value – min value**

**Syntax:**

```
ndarray.ptp(axis=None, out=None)
```

Sr. No.	Parameter	Description
1	arr	input array.

2	axis	axis along which we want the range value. Otherwise, it will consider arr to be flattened(works on all the axis). <b>axis = 0 means along the column and axis = 1 means working along the row.</b>
3	out	[ndarray, optional] Different array in which we want to place the result. The array must have same dimensions as expected output.
4	Return	Range of the array (a scalar value if axis is none) or array with range of values along specified axis.

**Example 1:**

```
import numpy as np
# 1D array
arr = [1, 2, 7, 20, np.nan]
print("arr : ", arr)
print("Range of arr : ", np.ptp(arr))

#1D array
arr = [1, 2, 7, 10, 16]
print("arr : ", arr)
print("Range of arr : ", np.ptp(arr))
```

**Output:**

```
arr : [1, 2, 7, 20, nan]
Range of arr : nan
arr : [1, 2, 7, 10, 16]
Range of arr : 15
```

**Example 2:**

```
import numpy as np
a = np.array([[3,7,5],[8,4,3],[2,4,9]])

print 'Our array is:'
print a
print '\n'

print 'Applying ptp() function:'
print np.ptp(a)
print '\n'

print 'Applying ptp() function along axis 1:'
print np.ptp(a, axis = 1)
print '\n'
```

```
print 'Applying ptp() function along axis 0:'  
print np.ptp(a, axis = 0)
```

**Output:**

Our array is:

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

Applying ptp() function:

```
7
```

Applying ptp() function along axis 1:

```
[4 5 7]
```

Applying ptp() function along axis 0:

```
[6 3 6]
```

**3. numpy.percentile():**

- Percentile (or a centile) is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall.
- numpy.percentile() function used to compute the nth percentile of the given data (array elements) along the specified axis.
- The function numpy.percentile() takes the following arguments.

**Syntax:**

numpy.percentile(input, q, axis)

Sr. No.	Parameter	Description
1	input	It is the input array.
2	q	It is the percentile (1-100) which is calculated of the array element.
3	axis	It is the axis along which the percentile is to be calculated.

**Example:**

```
import numpy as np  
a = np.array([[30,40,70],[80,20,10],[50,90,60]])
```

```
print 'Our array is:'  
print a  
print '\n'
```

```
print 'Applying percentile() function:'  
print np.percentile(a,50)  
print '\n'  
  
print 'Applying percentile() function along axis 1:'  
print np.percentile(a,50, axis = 1)  
print '\n'  
  
print 'Applying percentile() function along axis 0:'  
print np.percentile(a,50, axis = 0)
```

**Output:**

Our array is:

```
[[30 40 70]  
 [80 20 10]  
 [50 90 60]]
```

Applying percentile() function:  
50.0

Applying percentile() function along axis 1:  
[ 40. 20. 60.]

Applying percentile() function along axis 0:  
[ 50. 40. 60.]

**4. numpy.median():**

- Median is defined as the value that is used to separate the higher range of data sample with a lower range of data sample.
- Median is defined as the value separating the higher half of a data sample from the lower half.
- The function `numpy.median()` is used to calculate the median of the multi-dimensional or one-dimensional arrays.
- **How to calculate median?**
  1. Given data points.
  2. Arrange them in ascending order
  3. Median = middle term if total no. of terms are odd.
  4. Median = Average of the terms in the middle (if total no. of terms are even)

**Syntax:**

```
numpy.median(arr, axis = None)
```

Sr. No.	Parameter	Description
1	arr	It is the input array.
2	axis	[int or tuples of int]axis along which we want to calculate the median. Otherwise, it will consider arr to be flattened(works on all the axis). <b>axis = 0 means along the column and axis = 1 means working along the row.</b>

**Example 1:**

```
import numpy as np
```

```
# 1D array
```

```
arr = [20, 2, 7, 1, 34]
```

```
print("arr : ", arr)
```

```
print("median of arr : ", np.median(arr))
```

**Output:**

```
arr : [20, 2, 7, 1, 34]
```

```
median of arr : 7.0
```

**Example 2:**

```
import numpy as np
```

```
a = np.array([[30,65,70],[80,95,10],[50,90,60]])
```

```
print 'Our array is:'
```

```
print a
```

```
print '\n'
```

```
print 'Applying median() function:'
```

```
print np.median(a)
```

```
print '\n'
```

```
print 'Applying median() function along axis 0:'
```

```
print np.median(a, axis = 0)
```

```
print '\n'
```

```
print 'Applying median() function along axis 1:'
```

```
print np.median(a, axis = 1)
```

**Output:**

Our array is:

```
[[30 65 70]
```

```
[80 95 10]
```

```
[50 90 60]]
```

Applying median() function:

```
65.0
```

Applying median() function along axis 0:

```
[ 50. 90. 60.]
```

Applying median() function along axis 1:

```
[ 65. 80. 60.]
```

**5. numpy.mean():**

- The mean can be calculated by adding all the items of the arrays dividing by the number of array elements.
- We can also mention the axis along which the mean can be calculated.
- Arithmetic mean is the sum of elements along an axis divided by the number of elements.
- The numpy.mean() function returns the arithmetic mean of elements in the array.
- If the axis is mentioned, it is calculated along it.

**Example:**

```
import numpy as np
```

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

```
print 'Our array is:'
```

```
print a
```

```
print '\n'
```

```
print 'Applying mean() function:'
```

```
print np.mean(a)
```

```
print '\n'
```

```
print 'Applying mean() function along axis 0:'
```

```
print np.mean(a, axis = 0)
```

```
print '\n'
```

```
print 'Applying mean() function along axis 1:'
```

```
print np.mean(a, axis = 1)
```



**Output:**

Our array is:

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

```
[3 4 5]
```

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

Applying mean() function:

```
3.666666666667
```

Applying mean() function along axis 0:

```
[ 2.66666667 3.66666667 4.66666667]
```

Applying mean() function along axis 1:

```
[ 2. 4. 5.]
```

**6. numpy.average():**

- The numpy.average() function is used to find the weighted average along the axis of the multi-dimensional arrays where their weights are given in another array.
- Weighted average is an average resulting from the multiplication of each component by a factor reflecting its importance.
- The numpy.average() function computes the weighted average of elements in an array according to their respective weight given in another array.
- The function can have an axis parameter.
- If the axis is not specified, the array is flattened.
- Considering an array [1,2,3,4] and corresponding weights [4,3,2,1], the weighted average is calculated by adding the product of the corresponding elements and dividing the sum by the sum of weights.
- $\text{Weighted average} = (1*4+2*3+3*2+4*1)/(4+3+2+1)$

**Example 1:**

```
import numpy as np
```

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

```
print 'Our array is:'
```

```
print a
```

```
print '\n'
```

```
print 'Applying average() function:'
```

```
print np.average(a)
```

```
print '\n'
```

```
# this is same as mean when weight is not specified
wts = np.array([4,3,2,1])

print 'Applying average() function again:'
print np.average(a,weights = wts)
print '\n'

# Returns the sum of weights, if the returned parameter is set to True.
print 'Sum of weights'
print np.average([1,2,3, 4],weights = [4,3,2,1], returned = True)
```

**Output:**

Our array is:

[1 2 3 4]

Applying average() function:

2.5

Applying average() function again:

2.0

Sum of weights

(2.0, 10.0)

- In a multi-dimensional array, the axis for computation can be specified.

**Example 2:**

```
import numpy as np
a = np.arange(6).reshape(3,2)

print 'Our array is:'
print a
print '\n'

print 'Modified array:'
wt = np.array([3,5])
print np.average(a, axis = 1, weights = wt)
print '\n'
```

```
print 'Modified array:'  
print np.average(a, axis = 1, weights = wt, returned = True)
```

**Output:**

Our array is:

```
[[0 1]  
 [2 3]  
 [4 5]]
```

Modified array:

```
[ 0.625 2.625 4.625]
```

Modified array:

```
(array([ 0.625, 2.625, 4.625]), array([ 8., 8., 8.]))
```

**7. Standard Deviation:**

- Standard deviation is the square root of the average of squared deviations from mean.
- The formula for standard deviation is as follows:  
$$\text{std} = \sqrt{\text{mean}(\text{abs}(\mathbf{x} - \mathbf{x.mean()})^2)}$$
- If the array is [1, 2, 3, 4], then its mean is 2.5.
- Hence the squared deviations are [2.25, 0.25, 0.25, 2.25] and the square root of its mean divided by 4, i.e.,  $\sqrt{5/4}$  is 1.1180339887498949.

**Example:**

```
import numpy as np  
print np.std([1,2,3,4])
```

**Output:**

```
1.1180339887498949
```

**8. Variance:**

- Variance is the average of squared deviations, i.e.,  $\text{mean}(\text{abs}(\mathbf{x} - \mathbf{x.mean()})^2)$ .
- In other words, the standard deviation is the square root of variance.

**Example:**

```
import numpy as np  
print np.var([1,2,3,4])
```

**Output:**

```
1.25
```

**NumPy Sorting and Searching:**

- Numpy provides a variety of functions for sorting and searching.
- There are various sorting algorithms like quicksort, merge sort and heapsort which is implemented using the `numpy.sort()` function.
- The kind of the sorting algorithm to be used in the sort operation must be mentioned in the function call.
- Let's discuss the sorting algorithm which is implemented in `numpy.sort()`

SN	Algorithm	Worst case complexity
1	Quick Sort	$O(n^2)$
2	Merge Sort	$O(n \log(n))$
3	Heap Sort	$O(n \log(n))$

- The syntax to use the `numpy.sort()` function is given below.

**`numpy.sort(a, axis, kind, order)`**

- It accepts the following parameters.

Sr. No.	Parameter	Description
1	input	It represents the input array which is to be sorted.
2	axis	It represents the axis along which the array is to be sorted. If the axis is not mentioned, then the sorting is done along the last available axis.
3	kind	It represents the type of sorting algorithm which is to be used while sorting. The default is quick sort.
4	order	It represents the field according to which the array is to be sorted in the case if the array contains the fields.

**Example:**

```
import numpy as np
a = np.array([[10,2,3],[4,5,6],[7,8,9]])
print("Sorting along the columns:")
print(np.sort(a))
print("Sorting along the rows:")
print(np.sort(a, 0))
data_type = np.dtype([('name', 'S10'),('marks',int)])
arr = np.array([('Mukesh',200),('John',251)],dtype = data_type)
print("Sorting data ordered by name")
print(np.sort(arr,order = 'name'))
```

**Output:**

Sorting along the columns:

```
[[ 2  3 10]
```

```
 [ 4  5  6]
```

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

Sorting along the rows:

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

```
 [ 7  5  6]
```

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

Sorting data ordered by name

```
[(b'John', 251) (b'Mukesh', 200)]
```

**1. numpy.argsort() function:**

- This function is used to perform an indirect sort on an input array that is, it returns an array of indices of data which is used to construct the array of sorted data.

**Example:**

```
import numpy as np
a = np.array([90, 29, 89, 12])
print("Original array:\n",a)
sort_ind = np.argsort(a)
print("Printing indices of sorted data\n",sort_ind)
sort_a = a[sort_ind]
print("printing sorted array")
for i in sort_ind:
    print(a[i],end = " ")
```

**Output:**

Original array:

```
[90 29 89 12]
```

Printing indices of sorted data

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

printing sorted array

```
12 29 89 90
```

**2. numpy.lexsort() function:**

- This function is used to sort the array using the sequence of keys indirectly.
- This function performs similarly to the numpy.argsort() which returns the array of indices of sorted data.

**Example:**

```
import numpy as np
a = np.array(['a','b','c','d','e'])
```

```
b = np.array([12, 90, 380, 12, 211])
ind = np.lexsort((a,b))
print("printing indices of sorted data")
print(ind)
print("using the indices to sort the array")
for i in ind:
    print(a[i],b[i])
```

**Output:**

```
printing indices of sorted data
[0 3 1 4 2]
using the indices to sort the array
a 12
d 12
b 90
e 211
c 380
```

**3. numpy.nonzero() function:**

- This function is used to find the location of the non-zero elements from the array.

**Example:**

```
import numpy as np
b = np.array([12, 90, 380, 12, 211])
print("printing original array",b)
print("printing location of the non-zero elements")
print(b.nonzero())
```

**Output:**

```
printing original array [ 12  90 380  12 211]
printing location of the non-zero elements
(array([0, 1, 2, 3, 4]))
```

**4. numpy.where() function:**

- This function is used to return the indices of all the elements which satisfies a particular condition.

**Example:**

```
import numpy as np
b = np.array([12, 90, 380, 12, 211])
print(np.where(b>12))
c = np.array([[20, 24],[21, 23]])
print(np.where(c>20))
```

**Output:**

```
(array([1, 2, 4], dtype=int64),)
(array([0, 1, 1], dtype=int64), array([1, 0, 1], dtype=int64))
```

**NumPy Copies and Views:**

- The copy of an input array is physically stored at some other location and the content stored at that particular location is returned which is the copy of the input array whereas the different view of the same memory location is returned in the case of view.

**1. Array Assignment:**

- The assignment of a numpy array to another array doesn't make the direct copy of the original array, instead, it makes another array with the same content and same id.
- It represents the reference to the original array.
- Changes made on this reference are also reflected in the original array.
- The id() function returns the universal identifier of the array similar to the pointer in C.

**Example:**

```
import numpy as np
a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])
print("Original Array:\n",a)
print("\nID of array a:",id(a))
b = a
print("\nmaking copy of the array a")
print("\nID of b:",id(b))
b.shape = 4,3;
print("\nChanges on b also reflect to a:")
print(a)
```

**Output:**

Original Array:

```
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]
```

ID of array a: 139663602288640

making copy of the array a

ID of b: 139663602288640

Changes on b also reflect to a:

```
[[ 1  2  3]
 [ 4  9  0]
 [ 2  3  1]
 [ 2  3 19]]
```

**2. ndarray.view() method:**

- The ndarray.view() method returns the new array object which contains the same content as the original array does.
- Since it is a new array object, changes made on this object do not reflect the original array.

**Example:**

```
import numpy as np
a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])
print("Original Array:\n",a)
print("\nID of array a:",id(a))
b = a.view()
print("\nID of b:",id(b))
print("\nprinting the view b")
print(b)
b.shape = 4,3;
print("\nChanges made to the view b do not reflect a")
print("\nOriginal array \n",a)
print("\nview\n",b)
```

**Output:**

```
Original Array:
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]
ID of array a: 140280414447456
ID of b: 140280287000656
printing the view b
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]
Changes made to the view b do not reflect a
Original array
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]
View
[[ 1  2  3]
 [ 4  9  0]
 [ 2  3  1]
 [ 2  3 19]]
```



**3. ndarray.copy() method:**

- It returns the deep copy of the original array which doesn't share any memory with the original array.
- The modification made to the deep copy of the original array doesn't reflect the original array.

**Example:**

```
import numpy as np
a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])
print("Original Array:\n",a)
print("\nID of array a:",id(a))
b = a.copy()
print("\nID of b:",id(b))
print("\nprinting the deep copy b")
print(b)
b.shape = 4,3;
print("\nChanges made to the copy b do not reflect a")
print("\nOriginal array \n",a)
print("\nCopy\n",b)
```

**Output:**

```
Original Array:
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]
ID of array a: 139895697586176
ID of b: 139895570139296
printing the deep copy b
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]
Changes made to the copy b do not reflect a
Original array
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]
Copy
[[ 1  2  3]
 [ 4  9  0]
 [ 2  3  1]
 [ 2  3 19]]
```

**NumPy Matrix Library:**

- NumPy contains a matrix library, i.e. numpy.matlib which is used to configure matrices instead of ndarray objects.

**1. numpy.matlib.empty() function:**

- This function is used to return a new matrix with the uninitialized entries.
- The syntax to use this function is given below.

**numpy.matlib.empty(shape, dtype, order)**

- It accepts the following parameter.
  1. shape: It is the tuple defining the shape of the matrix.
  2. dtype: It is the data type of the matrix.
  3. order: It is the insertion order of the matrix, i.e. C or F.

**Example:**

```
import numpy as np
import numpy.matlib
print(numpy.matlib.empty((3,3)))
```

**Output:**

```
[[6.90262230e-310 6.90262230e-310 6.90262304e-310]
 [6.90262304e-310 6.90261674e-310 6.90261552e-310]
 [6.90261326e-310 6.90262311e-310 3.95252517e-322]]
```

**2. Numpy.matlib.zeros () function:**

- This function is used to create the matrix where the entries are initialized to zero.

**Example:**

```
import numpy as np
import numpy.matlib
print(numpy.matlib.zeros((4,3)))
```

**Output:**

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

**3. numpy.matlib.ones() function:**

- This function returns a matrix with all the elements initialized to 1.

**Example:**

```
import numpy as np
import numpy.matlib
print(numpy.matlib.ones((2,2)))
```

**Output:**

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

**4. numpy.matlib.eye() function:**

- This function returns a matrix with the diagonal elements initialized to 1 and zero elsewhere.
- The syntax to use this function is given below.

**numpy.matlib.eye(n, m, k, dtype)**

- It accepts the following parameters.
  1. n: It represents the number of rows in the resulting matrix.
  2. m: It represents the number of columns, defaults to n.
  3. k: It is the index of diagonal.
  4. dtype: It is the data type of the output

**Example:**

```
import numpy as np  
import numpy.matlib  
print(numpy.matlib.eye(n = 3, M = 3, k = 0, dtype = int))
```

**Output:**

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

**5. numpy.matlib.identity() function:**

- This function is used to return an identity matrix of the given size.
- An identity matrix is the one with diagonal elements initializes to 1 and all other elements to zero.

**Example:**

```
import numpy as np  
import numpy.matlib  
print(numpy.matlib.identity(5, dtype = int))
```

**Output:**

```
[[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]]
```

**6. numpy.matlib.rand() function:**

- This function is used to generate a matrix where all the entries are initialized with random values.

**Example:**

```
import numpy as np
import numpy.matlib
print(numpy.matlib.rand(3,3))
```

**Output:**

```
[[0.86201511 0.86980769 0.06704884]
 [0.80531086 0.53814098 0.84394673]
 [0.85653048 0.8146121 0.35744405]]
```

**NumPy Linear Algebra:**

- Numpy provides the following functions to perform the different algebraic calculations on the input data.

SN	Function	Definition
1	dot()	It is used to calculate the dot product of two arrays.
2	vdot()	It is used to calculate the dot product of two vectors.
3	inner()	It is used to calculate the inner product of two arrays.
4	matmul()	It is used to calculate the matrix multiplication of two arrays.
5	det()	It is used to calculate the determinant of a matrix.
6	solve()	It is used to solve the linear matrix equation.
7	inv()	It is used to calculate the multiplicative inverse of the matrix.

**1. numpy.dot() function:**

- This function is used to return the dot product of the two matrices.
- It is similar to the matrix multiplication.

**Example:**

```
import numpy as np
a = np.array([[100,200],[23,12]])
b = np.array([[10,20],[12,21]])
dot = np.dot(a,b)
print(dot)
```

**Output:**

```
[[3400 6200]
 [ 374  712]]
```

The dot product is calculated as:

```
[100 * 10 + 200 * 12, 100 * 20 + 200 * 21] [23*10+12*12, 23*20 + 12*21]
```

**2. numpy.vdot() function:**

- This function is used to calculate the dot product of two vectors.
- It can be defined as the sum of the product of corresponding elements of multi-dimensional arrays.

**Example:**

```
import numpy as np
a = np.array([[100,200],[23,12]])
```

```
b = np.array([[10,20],[12,21]])  
vdot = np.vdot(a,b)  
print(vdot)
```

**Output:**

5528

- $\text{np.vdot}(a,b) = 100 * 10 + 200 * 20 + 23 * 12 + 12 * 21 = 5528$

**3. numpy.inner() function:**

- This function returns the sum of the product of inner elements of the one-dimensional array.
- For n-dimensional arrays, it returns the sum of the product of elements over the last axis.

**Example:**

```
import numpy as np
a = np.array([1,2,3,4,5,6])
b = np.array([23,23,12,2,1,2])
inner = np.inner(a,b)
print(inner)
```

**Output:**

130

**4. numpy.matmul() function:**

- It is used to return the multiplication of the two matrices.
- It gives an error if the shape of both matrices is not aligned for multiplication.

**Example:**

```
import numpy as np
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
b = np.array([[23,23,12],[2,1,2],[7,8,9]])
mul = np.matmul(a,b)
print(mul)
```

**Output:**

```
[[ 48 49 43]
 [144 145 112]
 [240 241 181]]
```

**5. numpy determinant:**

- The determinant of the matrix can be calculated using the diagonal elements. The determinant of following 2 X 2 matrix

A	B
C	D

can be calculated as  $AD - BC$ .

- The `numpy.linalg.det()` function is used to calculate the determinant of the matrix.

**Example:**

```
import numpy as np
a = np.array([[1,2],[3,4]])
print(np.linalg.det(a))
```

**Output:**

-2.0000000000000004

#### 6. **numpy.linalg.solve() function:**

- This function is used to solve a quadratic equation where values can be given in the form of the matrix. The following linear equations

$$3X + 2Y + Z = 10$$

$$X + Y + Z = 5$$

can be represented by using three matrices as:

$$\begin{matrix} 3 & 2 & 1 \\ 1 & 1 & 1 \end{matrix}$$

$$\begin{matrix} X & Y & Z \end{matrix}$$

$$\text{and } \begin{matrix} 10 & 5 \end{matrix}$$

- The two matrices can be passed into the `numpy.solve()` function given as follows.

#### **Example:**

```
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[1,2],[3,4]])
print(np.linalg.solve(a, b))
```

#### **Output:**

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

#### 7. **numpy.linalg.inv() function:**

- This function is used to calculate the multiplicative inverse of the input matrix.

#### **Example:**

```
import numpy as np
a = np.array([[1,2],[3,4]])
print("Original array:\n",a)
b = np.linalg.inv(a)
print("Inverse:\n",b)
```

#### **Output:**

Original array:

```
[[1 2]
```

```
[3 4]]
```

Inverse:

```
[[ -2.  1.]
```

```
[ 1.5 -0.5]]
```