



NumPy Library

What is NumPy?



- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
- NumPy stands for Numerical Python.
-

Why Use NumPy?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called `ndarray` , it provides a lot of supporting functions that make working with `ndarray` very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.



Why is NumPy Faster Than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behavior is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

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++.

Installation of NumPy

- 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
```




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

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

NumPy as np

NumPy is usually imported under the **np** alias.

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)
```

Checking NumPy Version

The version string is stored under `__version__` attribute.

Example

```
import numpy as np  
print(np.__version__)
```



NumPy Creating Arrays

Create a NumPy ndarray Object

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.

Example

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

- 
- To create an `ndarray`, we can pass a list, tuple or any array-like object into the `array()` method, and it will be converted into an `ndarray`:

Example

Use a tuple to create a NumPy array:

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

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

1) 0-D Arrays :

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example

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

2) 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

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

3) 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)
```

4) 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)
```

Iterate through loop :

```
for i in arr:
    for j in i:
        for k in j:
            print(k)
```


Check Number of Dimensions?

NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

Example

```
import numpy as np
```

```
a = np.array(42)
```

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

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

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

```
print(a.ndim)
```

```
print(b.ndim)
```

```
print(c.ndim)
```

```
print(d.ndim)
```

Higher Dimensional Arrays

- An array can have any number of dimensions.
- When the array is created, you can define the number of dimensions by using the `ndmin` argument..

Example

Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

- In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

NumPy Array Indexing

- Array indexing is the same as accessing an array element.
- You can access an array element by referring to its index number.
- The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

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

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Example

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st dim: ', arr[0, 1])
```

Access 3-D Arrays

Example

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

Negative Indexing

Use negative indexing to access an array from the end.

Example

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

NumPy Array Slicing

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: `[start:end]`

We can also define the step, like this: `[start:end:step]`

If we don't pass **start** its considered **0**


If we don't pass **end** its considered **length of array** in that dimension

If we don't pass **step** its considered **1**

Example

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Note: The result *includes* the start index, but *excludes* the end index.



```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

Negative Slicing

Example

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```



STEP

Use the **step** value to determine the step of the slicing:

Example

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

Return every other element from the entire array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:, :2])
```

Slicing 2-D Arrays

Example

From the second element, slice elements from index 1 to index 4 (not included):

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

From both elements, return index 2:

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

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

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



NumPy Data Types

By default Python have these data types:

strings

integer


float

boolean

Complex

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.

i – integer

b – boolena

u – unsigned integer

f – float

c – complex float

m – timedelta

M – datetime

O – object

S – string

U – Unicode string

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

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

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


Creating Arrays With a Defined Data Type

We use the `array()` function to create arrays, this function can take an optional argument: `dtype` that allows us to define the expected data type of the array elements:

Example

Create an array with data type string:

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

Output :

```
array([b'1', b'2', b'3', b'4'], dtype='<S1')
```

Note : `'b'` is known as a "bytes" string.

For **i**, **u**, **f**, **S** and **U** we can define size as well.

Example

Create an array with data type 4 bytes integer:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='i4')
print(arr)
print(arr.dtype)
```

What if a Value Can Not Be Converted?

If a type is given in which elements can't be casted then NumPy will raise a `ValueError`.

A non integer string like 'a' can not be converted to integer (will raise an error):

```
import numpy as np
arr = np.array(['a', '2', '3'], dtype='i')
```

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)
```


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)
```

Change data type from integer to boolean:

```
import numpy as np
arr = np.array([1, 0, 3])
newarr = arr.astype(bool)
print(newarr)
print(newarr.dtype)
```



NumPy Array Copy vs View

- The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
 - The copy *owns* the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.
 - The view *does not own* the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.
- 



COPY:

Example

Make a copy, change the original array, and display both arrays:

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
x = arr.copy()  
arr[0] = 42  
  
print(arr)  
print(x)
```




VIEW:

Example

Make a view, change the original array, and display both arrays:

```
import numpy as np
```

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

```
x = arr.view()
```

```
arr[0] = 42
```

```
print(arr)
```

```
print(x)
```

NumPy Array Shape

The shape of an array is the number of elements in each dimension.

Get the Shape of an Array

NumPy arrays have an attribute called **shape** that returns a tuple with each index having the number of corresponding elements.

Example

```
import numpy as np  
  
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
  
print(arr.shape)
```

The example above returns **(2, 4)** , which means that the array has 2 dimensions, and each dimension has 4 elements.

Example

Create an array with 5 dimensions using `ndmin` using a vector with values 1,2,3,4 and verify that last dimension has value 4:

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('shape of array :', arr.shape)
```

NumPy Array Reshaping

- Reshaping means changing the shape of an array.
- The shape of an array is the number of elements in each dimension.
- By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Example

- Convert the following 1-D array with 12 elements into a 2-D array.
- The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

```
newarr = arr.reshape(4, 3)
```

```
print(newarr)
```

Reshape From 1-D to 3-D

Example

- Convert the following 1-D array with 12 elements into a 3-D array.
- The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

```
newarr = arr.reshape(2, 3, 2)
```

```
print(newarr)
```

Can We Reshape Into any Shape?

- Yes, as long as the elements required for reshaping are equal in both shapes.
- We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require $3 \times 3 = 9$ elements.

Example

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
newarr = arr.reshape(3, 3)
```

```
print(newarr)
```

Unknown Dimension

- ▶ You are allowed to have one "unknown" dimension.
- ▶ Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.
- ▶ Pass **-1** as the value, and NumPy will calculate this number for you.

Example

Convert 1D array with 8 elements to 3D array with 2x2 elements:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
newarr = arr.reshape(2, 2, -1)
```

```
print(newarr)
```

Note: We can not pass **-1** to more than one dimension.

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array. We can use `reshape(-1)` to do this.

Example

Convert the array into a 1D array:

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

Joining NumPy Arrays

- Joining means putting contents of two or more arrays in a single array.
- In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.
- We pass a sequence of arrays that we want to join to the `concatenate()` function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Example

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

Example

Join two 2-D arrays along rows (axis=1):

```
import numpy as np
```

```
arr1 = np.array([[1, 2], [3, 4]])
```

```
arr2 = np.array([[5, 6], [7, 8]])
```

```
arr = np.concatenate((arr1, arr2), axis=1)
```

```
print(arr)
```

Splitting NumPy Arrays


- Splitting is reverse operation of Joining.
- Joining merges multiple arrays into one and Splitting breaks one array into multiple.
- We use `array_split()` for splitting arrays, we pass it the array we want to split and the number of splits.

Example

Split the array in 3 parts:

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

Note: The return value is an array containing three arrays.



If the array has less elements than required, it will adjust from the end accordingly.

Example

Split the array in 4 parts:

```
import numpy as np
```

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

```
newarr = np.array_split(arr, 4)
```

```
print(newarr)
```

Split Into Arrays

- The return value of the `array_split()` method is an array containing each of the split as an array.
- If you split an array into 3 arrays, you can access them from the result just like any array element:

Example

```
import numpy as np
```

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

```
newarr = np.array_split(arr, 3)
```

```
print(newarr[0])
```

```
print(newarr[1])
```

```
print(newarr[2])
```

Splitting 2-D Arrays

Use the same syntax when splitting 2-D arrays.

Use the `array_split()` method, pass in the array you want to split and the number of splits you want to do.

Example

Split the 2-D array into three 2-D arrays.

```
import numpy as np
```

```
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
```

```
newarr = np.array_split(arr, 3)
```

```
print(newarr)
```


Example

Split the 2-D array into three 2-D arrays.

```
import numpy as np
```

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],  
               [16, 17, 18]])
```

```
newarr = np.array_split(arr, 3)
```

```
print(newarr)
```

Output:

```
[array([[1, 2, 3], [4, 5, 6]]), array([[ 7, 8, 9], [10, 11, 12]]), array([[13, 14,  
15], [16, 17, 18]])]
```

- you can specify which axis you want to do the split around.
- The example below also returns three 2-D arrays, but they are split along the row (axis=1).

Example

Split the 2-D array into three 2-D arrays along rows.

```
import numpy as np
```

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],  
[16, 17, 18]])
```

```
newarr = np.array_split(arr, 3, axis=1)
```

```
print(newarr)
```

Searching Arrays

- You can search an array for a certain value, and return the indexes that get a match.
- To search an array, use the `where()` method.

Example

Find the indexes where the value is 4:

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

The example above will return a tuple: `(array([3, 5, 6]),)`

Which means that the value 4 is present at index 3, 5, and 6.

Example

Find the indexes where the values are even:

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
x = np.where(arr%2 == 0)  
  
print(x)
```

Find the indexes where the values are odd:

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
x = np.where(arr%2 == 1)  
  
print(x)
```

Search Sorted

There is a method called `searchsorted()` which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

The `searchsorted()` method is assumed to be used on sorted arrays.

Example

Find the indexes where the value 7 should be inserted:

```
import numpy as np  
  
arr = np.array([6, 7, 8, 9])  
  
x = np.searchsorted(arr, 7)  
  
print(x)
```

Search From the Right Side

By default the left most index is returned, but we can give `side='right'` to return the right most index instead.

Example

Find the indexes where the value 7 should be inserted, starting from the right:


```
import numpy as np  
arr = np.array([6, 7, 8, 9])  
x = np.searchsorted(arr, 7, side='right')  
print(x)
```

Sorting Arrays

- Sorting means putting elements in an *ordered sequence*.
- *Ordered sequence* is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
- The NumPy ndarray object has a function called `sort()` , that will sort a specified array.

Example

```
import numpy as np  
  
arr = np.array([3, 2, 0, 1])  
  
print(np.sort(arr))
```

You can also sort arrays of strings, or any other data type:

Example

Sort the array alphabetically:

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

Sorting a 2-D Array

If you use the `sort()` method on a 2-D array, both arrays will be sorted:

Example

Sort a 2-D array:

```
import numpy as np
```

```
arr = np.array([[3, 2, 4], [5, 0, 1]])
```

```
print(np.sort(arr))
```

NumPy - Array Creation Routines

A new **ndarray** object can be constructed by any of the following array creation routines or using a low-level ndarray constructor.

[numpy.empty](#)

It creates an uninitialized array of specified shape and dtype. It uses the following constructor

```
numpy.empty(shape, dtype = float, order = 'C')
```

The constructor takes the following parameters.

Sr.No.	Parameter & Description
1	Shape Shape of an empty array in int or tuple of int
2	Dtype Desired output data type. Optional
3	Order 'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example

The following code shows an example of an empty array.

```
import numpy as np
x = np.empty([3,2], dtype = int)
print x
```

The output is as follows –

```
[[22649312 1701344351]
 [1818321759 1885959276]
 [16779776 156368896]]
```

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



numpy.zeros

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

`numpy.zeros(shape, dtype = float, order = 'C')`

Example

array of five zeros. Default dtype is float

```
import numpy as np
```

```
x = np.zeros(5)
```

```
print x
```

The output is as follows –

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



Example

```
import numpy as np
```

```
x = np.zeros((5,), dtype = int)
```

```
print x
```

Now, the output would be as follows

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

numpy.ones

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

`numpy.ones(shape, dtype = None, order = 'C')`

Example

array of five ones. Default dtype is float

```
import numpy as np
```

```
x = np.ones(5) print x
```

The output is as follows

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

Example

```
import numpy as np
```

```
x = np.ones([2,2], dtype = int)
```

```
print x
```

Now, the output would be as follows

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


NumPy - Broadcasting

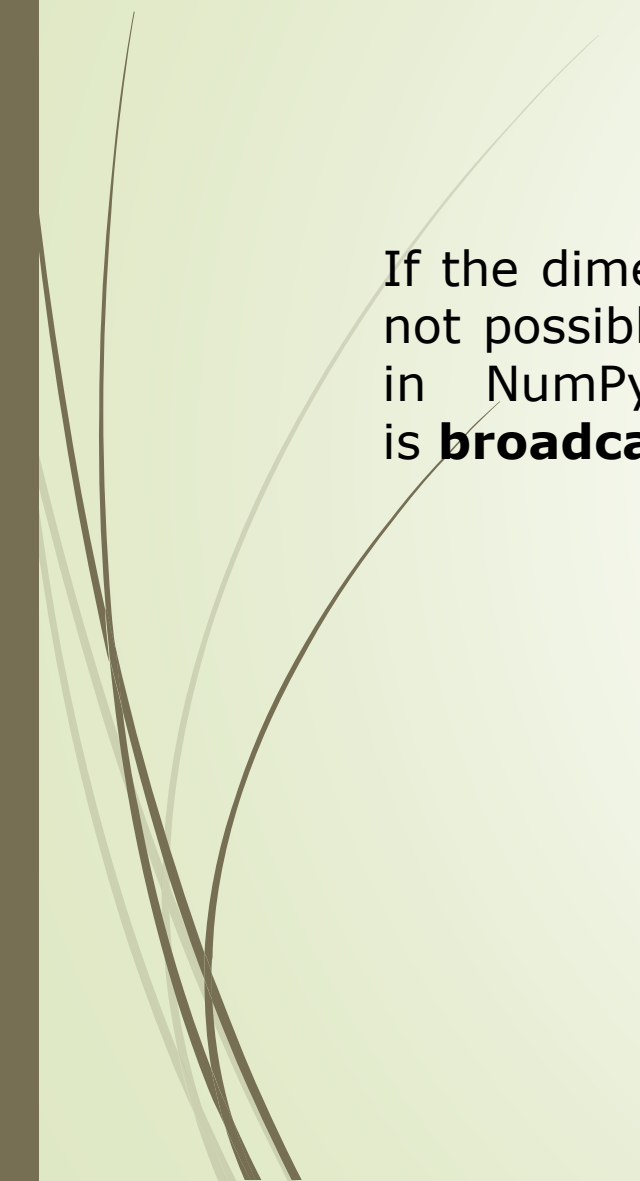

The term **broadcasting** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements. If two arrays are of exactly the same shape, then these operations are smoothly performed.

Example

```
import numpy as np
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
c = a * b
print c
```

Its output is as follows

```
[10 40 90 160]
```



If the dimensions of two arrays are dissimilar, element-to-element operations are not possible. However, operations on arrays of non-similar shapes is still possible in NumPy, because of the broadcasting capability. The smaller array is **broadcast** to the size of the larger array so that they have compatible shapes.

Example

```
import numpy as np

a =
np.array([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],[30.0,30.0,3
0.0]])

b = np.array([1.0,2.0,3.0])

print 'First array:'
print a
print '\n'
print 'Second array:'
print b
print '\n'
print 'First Array + Second Array'
print a + b
```



The output of this program would be as follows

First array:

```
[[ 0.  0.  0.] [ 10. 10. 10.] [ 20. 20. 20.] [ 30. 30. 30.]]
```

Second array:

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

First Array + Second Array

```
[[ 1.  2.  3.] [ 11. 12. 13.] [ 21. 22. 23.] [ 31. 32. 33.]]
```

NumPy - Array From Existing Data

[numpy.asarray](#)

This function is similar to `numpy.array` except for the fact that it has fewer parameters. This routine is useful for converting Python sequence into ndarray.

```
numpy.asarray(a, dtype = None, order = None)
```

Sr.No.	Parameter & Description
1	a Input data in any form such as list, list of tuples, tuples, tuple of tuples or tuple of lists
2	dtype By default, the data type of input data is applied to the resultant ndarray
3	order C (row major) or F (column major). C is default



Example

convert list to ndarray

```
import numpy as np
```

```
x = [1,2,3]
```

```
a = np.asarray(x)
```

```
print a
```

```
import numpy as np
```

```
x = [1,2,3]
```

```
a = np.asarray(x, dtype = float)
```

```
print a
```



ndarray from tuple

import numpy as np

x = (1,2,3)

a = np.asarray(x)

print a

ndarray from list of tuples

import numpy as np

x = [(1,2,3),(4,5)]

a = np.asarray(x)

print a

numpy.fromiter

This function builds an **ndarray** object from any iterable object. A new one-dimensional array is returned by this function.

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

Sr.No.	Parameter & Description
1	iterable Any iterable object
2	dtype Data type of resultant array
3	count The number of items to be read from iterator. Default is -1 which means all data to be read

The following examples show how to use the built-in **range()** function to return a list object. An iterator of this list is used to form an **ndarray** object.



Example

```
# create list object using range function
```

```
import numpy as np
```

```
list = range(5)
```

```
print list
```

```
# obtain iterator object from list
```

```
import numpy as np
```

```
list = range(5)
```

```
it = iter(list)
```

```
# use iterator to create ndarray
```

```
x = np.fromiter(it, dtype = float)
```

```
print x
```

numpy.arange

This function returns an **ndarray** object containing evenly spaced values within a given range. The format of the function is as follows.

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

Sr.No.	Parameter & Description
1	start The start of an interval. If omitted, defaults to 0
2	stop The end of an interval (not including this number)
3	step Spacing between values, default is 1
4	dtype Data type of resulting ndarray. If not given, data type of input is used

Example

```
import numpy as np
x = np.arange(5)
print x
```

```
import numpy as np
# dtype set
x = np.arange(5, dtype = float)
print x
```

```
# start and stop parameters set
import numpy as np
x = np.arange(10, 20, 2)
print x
```

numpy.linspace

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

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

Sr.No.	Parameter & Description
1	start The starting value of the sequence
2	stop The end value of the sequence, included in the sequence if endpoint set to true
3	num The number of evenly spaced samples to be generated. Default is 50
4	endpoint True by default, hence the stop value is included in the sequence. If false, it is not included
5	retstep If true, returns samples and step between the consecutive numbers
6	dtype Data type of output ndarray

Example

```
import numpy as np
x = np.linspace(10,20,5)
print x
```

```
# endpoint set to false
import numpy as np
x = np.linspace(10,20, 5, endpoint = False)
print x
```

```
# find retstep value
import numpy as np
x = np.linspace(1,2,5, retstep = True)
print x
# retstep here is 0.25
```

Transpose

[numpy.transpose](#)

This function permutes the dimension of the given array.

The function takes the following parameters.

```
numpy.transpose(arr)
```

Example

```
import numpy as np
a = np.arange(12).reshape(3,4)
print 'The original array is:'
print a
print '\n'
print 'The transposed array is:'
print np.transpose(a)
```


Joining array (Continue.....)

[numpy.stack](#)

This function joins the sequence of arrays along a new axis.

```
numpy.stack(arrays, axis)
```

Sr.N o.	Parameter & Description
1	arrays Sequence of arrays of the same shape
2	axis Axis in the resultant array along which the input arrays are stacked

Example

```
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
print np.stack((a,b),0)
print np.stack((a,b),1)
```

Output:

Stack the two arrays along axis 0:

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

Stack the two arrays along axis 1:

```
[[[1 2]
 [5 6]]

 [[3 4]
 [7 8]]]
```

numpy.hstack

Variants of numpy.stack function to stack so as to make a single array horizontally.

Example

```
a = np.array([[1,2],[3,4]])  
b = np.array([[5,6],[7,8]])  
print np.hstack((a,b))
```

Output :

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

numpy.vstack

Variants of numpy.stack function to stack so as to make a single array vertically.

Example

```
a = np.array([[1, 2], [3, 4]])  
b = np.array([[5, 6], [7, 8]])  
print np.vstack((a, b))
```

Output :

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

Splitting an array (Continue....)

[numpy.hsplit](#)

The `numpy.hsplit` is a special case of `split()` function where axis is 1 indicating a horizontal split regardless of the dimension of the input array.

Example

```
import numpy as np
a = np.arange(16).reshape(4, 4)
print a
b = np.hsplit(a, 2)
```



Output :

First array:

```
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]
 [12 13 14 15]]
```

Horizontal splitting:

```
[array([[ 0, 1],
 [ 4, 5],
 [ 8, 9],
 [12, 13]]), array([[ 2, 3],
 [ 6, 7],
 [10, 11],
 [14, 15]])]
```

numpy.vsplit

numpy.vsplit is a special case of split() function where axis is 1 indicating a vertical split regardless of the dimension of the input array.

Example

```
import numpy as np
a = np.arange(16).reshape(4, 4)
print a
b = np.vsplit(a, 2)
```




Output :

First array:

```
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]
 [12 13 14 15]]
```

Vertical splitting:

```
[array([[0, 1, 2, 3],
 [4, 5, 6, 7]]), array([[ 8, 9, 10, 11],
 [12, 13, 14, 15]])]
```



NumPy - Mathematical Functions

NumPy provides standard trigonometric functions, functions for arithmetic operations, handling complex numbers, etc.

Trigonometric Functions

NumPy has standard trigonometric functions which return trigonometric ratios for a given angle in radians.


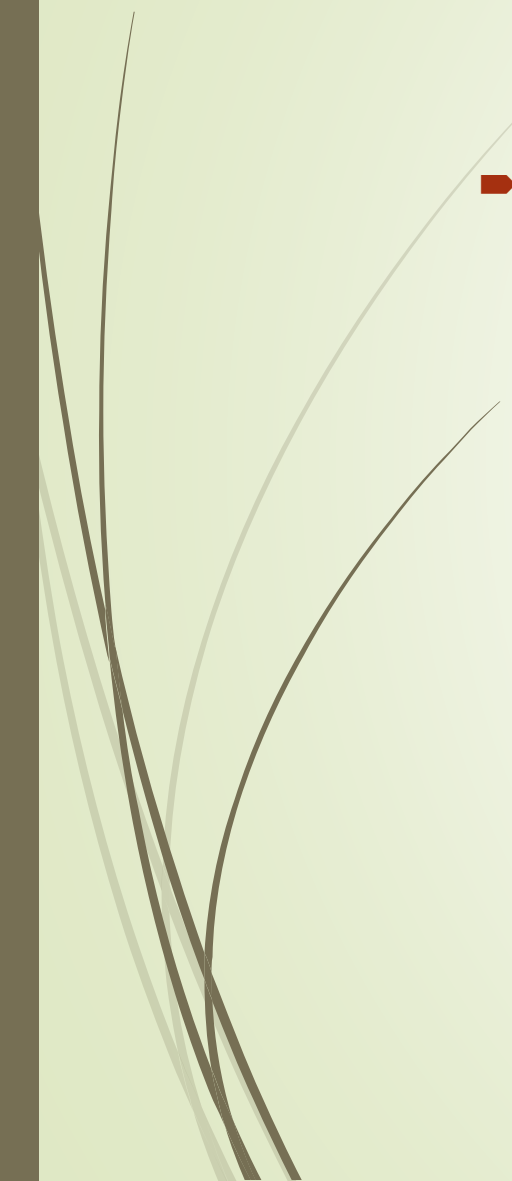
Example


```
import numpy as np
a = np.array([0, 30, 45, 60, 90])

print 'Sine of different angles:'
# Convert to radians by multiplying with pi/180
sin = np.sin(a*np.pi/180)

print 'Cosine values for angles in array:'
cos = np.cos(a*np.pi/180)

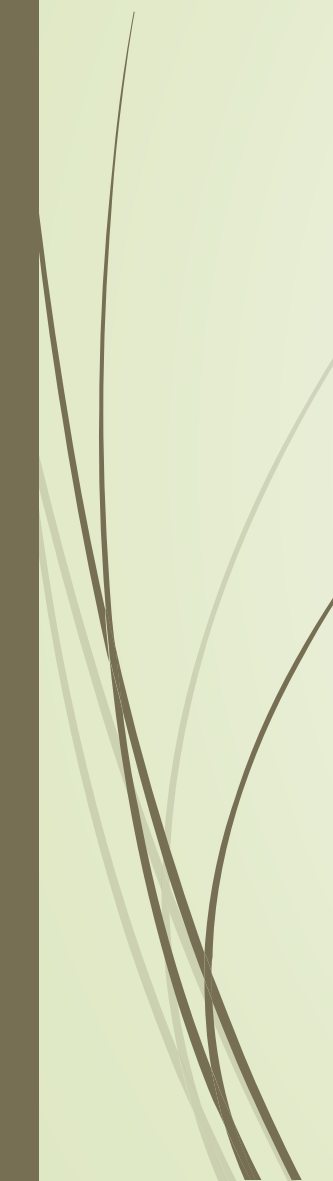

print 'Tangent values for given angles:'
tan = np.tan(a*np.pi/180)
```

- 
- 
- **arcsin**, **arccos**, and **arctan** functions return the trigonometric inverse of sin, cos, and tan of the given angle. The result of these functions can be verified by **numpy.degrees()** function by converting radians to degrees.



```
import numpy as np
a = np.array([0, 30, 45, 60, 90])
print 'Array containing sine values:'
sin = np.sin(a*np.pi/180)
print sin
print '\n'

print 'Compute sine inverse of angles. Returned
values are in radians.'
inv = np.arcsin(sin)
print inv print '\n'
print 'Check result by converting to degrees:'
print np.degrees(inv)
```



```
cos = np.cos(a*np.pi/180)
print cos
print 'Inverse of cos:'
inv = np.arccos(cos)
print inv

print 'In degrees:'
print np.degrees(inv)

print 'Tan function:'
tan = np.tan(a*np.pi/180)
print tan

print 'Inverse of tan:'
inv = np.arctan(tan)
print inv

print 'In degrees:' print np.degrees(inv)
```

Array containing sine values: [0. 0.5 0.70710678 0.8660254 1.]

Compute sine inverse of angles. Returned values are in radians.

[0. 0.52359878 0.78539816 1.04719755 1.57079633]

Check result by converting to degrees: [0. 30. 45. 60. 90.]

arccos and arctan functions behave similarly:

[1.000000000e+00 8.66025404e-01 7.07106781e-01 5.000000000e-01
6.12323400e-17]

Inverse of cos: [0. 0.52359878 0.78539816 1.04719755 1.57079633]

In degrees: [0. 30. 45. 60. 90.]

Tan function: [0.000000000e+00 5.77350269e-01 1.000000000e+00
1.73205081e+00 1.63312394e+16]

Inverse of tan: [0. 0.52359878 0.78539816 1.04719755 1.57079633]

In degrees: [0. 30. 45. 60. 90.]

Functions for Rounding

[numpy.around\(\)](#)

This is a function that returns the value rounded to the desired precision. The function takes the following parameters.

```
numpy.around(a, decimals)
```

Sr.No.	Parameter & Description
1	a Input data
2	decimals The number of decimals to round to. Default is 0. If negative, the integer is rounded to position to the left of the decimal point

Example

```
import numpy as np
a = np.array([1.0, 5.55, 123, 0.567, 25.532])
print 'Original array:'
print a
print '\n'
print 'After rounding:'
print np.around(a)
print np.around(a, decimals = 1)
print np.around(a, decimals = -1)
```

```
Original array: [ 1.  5.55 123.  0.567 25.532]
After rounding:
[ 1.  6. 123.  1. 26. ]
[ 1.  5.6 123.  0.6 25.5]
[ 0. 10. 120.  0. 30. ]
```

numpy.floor()

This function returns the largest integer not greater than the input parameter. The floor of the **scalar x** is the largest **integer i**, such that $i \leq x$. Note that in Python, flooring always is rounded away from 0.

Example

```
import numpy as np
a = np.array([-1.7, 1.5, -0.2, 0.6, 10])
print 'The given array:'
print a
print '\n'
print 'The modified array:'
print np.floor(a)
```

```
The given array:
[ -1.7  1.5 -0.2  0.6 10. ]
The modified array:
[ -2.  1. -1.  0. 10.]
```

numpy.ceil()

The `ceil()` function returns the ceiling of an input value, i.e. the ceil of the **scalar x** is the smallest **integer i**, such that **i >= x**.

Example

```
import numpy as np
a = np.array([-1.7, 1.5, -0.2, 0.6, 10])
print 'The given array:'
print a
print '\n'
print 'The modified array:'
print np.ceil(a)
```

```
The given array:
[ -1.7  1.5 -0.2  0.6 10. ]
The modified array:
[ -1.  2. -0.  1. 10.]
```

NumPy - Arithmetic Operations


Input arrays for performing arithmetic operations such as `add()`, `subtract()`, `multiply()`, and `divide()` must be either of the same shape or should conform to array broadcasting rules.

Example

```
import numpy as np
a = np.arange(9, dtype = np.float_).reshape(3,3)
print 'First array:'
print a
print '\n'

print 'Second array:'
b = np.array([10,10,10])
print b
print '\n'

print 'Add the two arrays:'
print np.add(a,b)
print '\n'
```



```
print 'Subtract the two arrays:'
print np.subtract(a,b)
print '\n'

print 'Multiply the two arrays:'
print np.multiply(a,b)
print '\n'

print 'Divide the two arrays:'
print np.divide(a,b)
```

It will produce the following output –

```
First array:
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
```

```
Second array:
[10 10 10]
```



Add the two arrays:

```
[[ 10. 11. 12.]  
 [ 13. 14. 15.]  
 [ 16. 17. 18.]]
```

Subtract the two arrays:

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

Multiply the two arrays:

```
[[  0. 10. 20.]  
 [ 30. 40. 50.]  
 [ 60. 70. 80.]]
```

Divide the two arrays:

```
[[ 0. 0.1 0.2]  
 [ 0.3 0.4 0.5]  
 [ 0.6 0.7 0.8]]
```


[numpy.reciprocal\(\)](#)

This function returns the reciprocal of argument, element-wise. For elements with absolute values larger than 100, the result is always 0 because of the way in which Python handles integer division. For integer 0, an overflow warning is issued.

Example

```
import numpy as np
a = np.array([0.25, 1.33, 1, 0, 100])
print 'Our array is:'
print a
print '\n'

print 'After applying reciprocal function:'
print np.reciprocal(a)
print '\n'

b = np.array([100], dtype = int)
print 'The second array is:'
print b
print '\n'

print 'After applying reciprocal function:'
print np.reciprocal(b)
```

It will produce the following output –

Our array is:

```
[ 0.25  1.33  1.  0. 100. ]
```

After applying reciprocal function:

```
main.py:9: RuntimeWarning: divide by zero encountered in reciprocal
```

```
print np.reciprocal(a)
```

```
[ 4.  0.7518797  1.  inf  0.01 ]
```

The second array is:

```
[100]
```

After applying reciprocal function:

```
[0]
```

numpy.power()

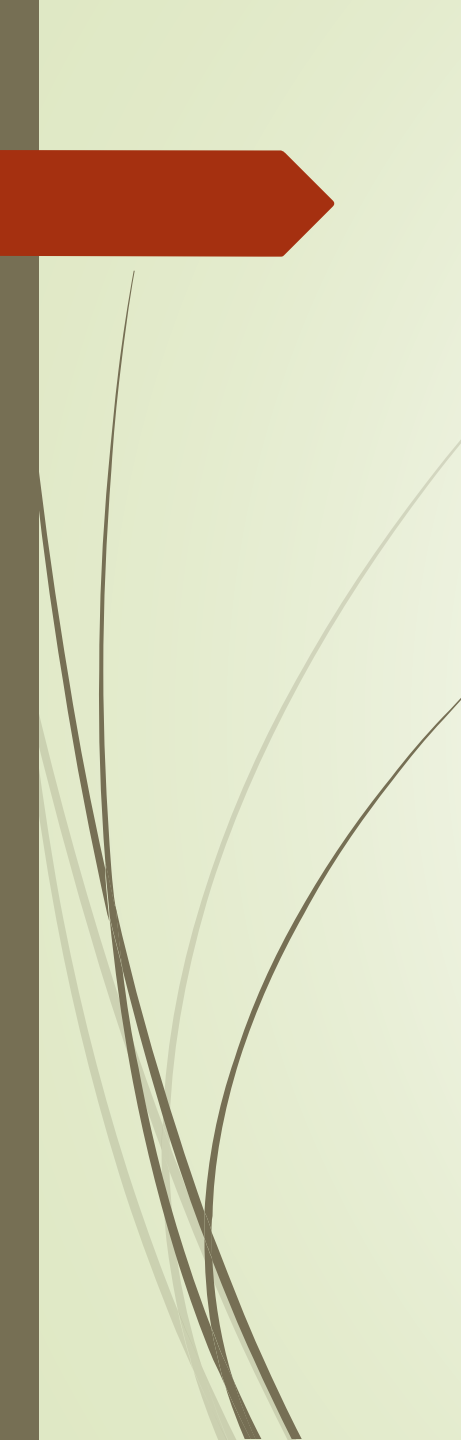
This function treats elements in the first input array as base and returns it raised to the power of the corresponding element in the second input array.

```
import numpy as np
a = np.array([10,100,1000])
print 'Our array is:'
print a
print '\n'

print 'Applying power function:'
print np.power(a,2)
print '\n'

print 'Second array:'
b = np.array([1,2,3])
print b
print '\n'

print 'Applying power function again:'
print np.power(a,b)
```



Our array is:

```
[ 10 100 1000]
```

Applying power function:

```
[ 100 10000 1000000]
```

Second array:

```
[1 2 3]
```

Applying power function again:

```
[ 10 10000 1000000000]
```

numpy.mod()


This function returns the remainder of division of the corresponding elements in the input array. The function **numpy.reminder()** also produces the same result.

```
import numpy as np
a = np.array([10,20,30])
b = np.array([3,5,7])
print 'First array:'
print a
print '\n'

print 'Second array:'
print b
print '\n'

print 'Applying mod() function:'
print np.mod(a,b)
print '\n'

print 'Applying remainder() function:'
print np.reminder(a,b)
```



First array:

[10 20 30]

Second array:

[3 5 7]

Applying mod() function:

[1 0 2]

Applying remainder() function:

[1 0 2]




NumPy - Statistical Functions

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 –

[numpy.amin\(\) and numpy.amax\(\)](#)

These functions return the minimum and the maximum from the elements in the given array along the specified axis.

Note : “**axis 0**” represents rows and “**axis 1**” represents columns.



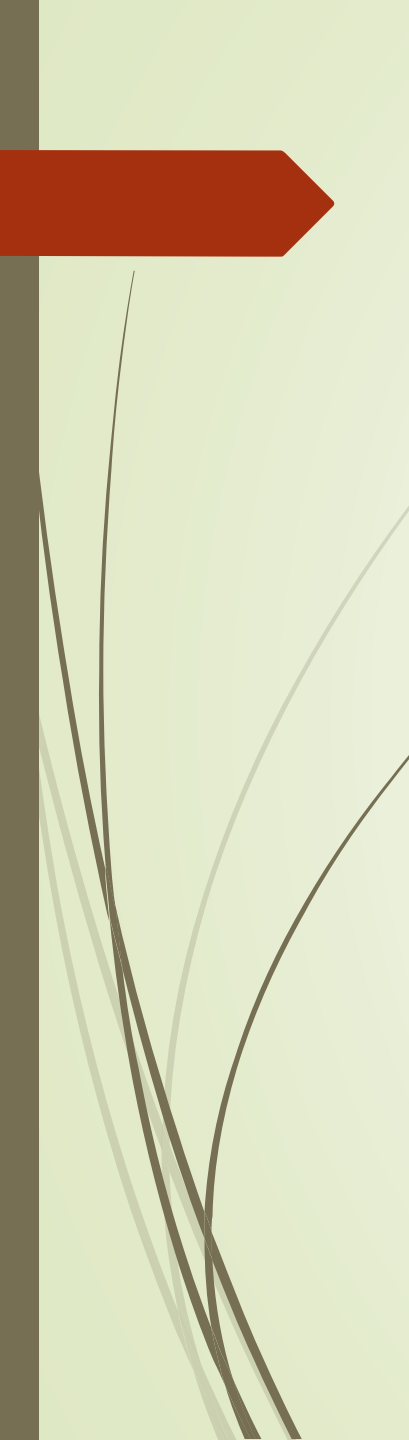
```
import numpy as np
a = np.array([[3,7,5],[8,4,3],[2,4,9]])
print 'Our array is:'
print a
print '\n'

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)
```



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]
```

numpy.median()

Median is defined as the value separating the higher half of a data sample from the lower half.


Example

```
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)
```



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.]

numpy.mean()


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.

```
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)
```



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.]
```


[numpy.average\(\)](#)

- 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.
- Weighted average = $(1*4+2*3+3*2+4*1)/(4+3+2+1)$


```
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)
```



Our array is:

[1 2 3 4]

Applying average() function:

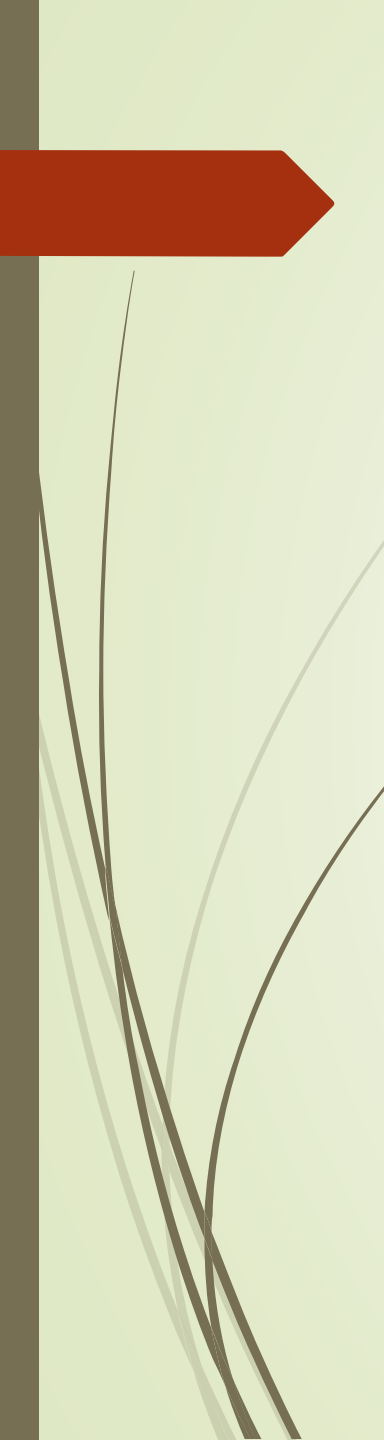
2.5

Applying average() function again:

2.0

Sum of weights


(2.0, 10.0)



```
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)
```



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.]))
```

Standard Deviation

Standard deviation is the square root of the average of squared deviations from mean. The formula for standard deviation is as follows –

```
std = sqrt(mean(abs(x - 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])
```

It will produce the following output –

```
1.1180339887498949
```



Variance

Variance is the average of squared deviations, i.e., **`mean(abs(x - 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])
```

It will produce the following output –

1.25

NumPy - Matrix Library

NumPy package contains a Matrix library **numpy.matlib**. This module has functions that return matrices instead of ndarray objects.

[matlib.empty\(\)](#)

The **matlib.empty()** function returns a new matrix without initializing the entries. The function takes the following parameters.

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

Sr.N o.	Parameter & Description
1	shape int or tuple of int defining the shape of the new matrix
2	Dtype Optional. Data type of the output
3	order C or F



Example

```
import numpy.matlib
import numpy as np
print np.matlib.empty((2,2))
# filled with random data
```

Output:

```
[[ 2.12199579e-314,  4.24399158e-314]
 [ 4.24399158e-314,  2.12199579e-314]]
```



`numpy.matlib.zeros()`

This function returns the matrix filled with zeros.

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

It will produce the following output –

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



`numpy.matlib.ones()`

This function returns the matrix filled with 1s.

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

It will produce the following output –

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

[numpy.matlib.eye\(\)](#)

This function returns a matrix with 1 along the diagonal elements and the zeros elsewhere. The function takes the following parameters.

```
numpy.matlib.eye(n, M, k, dtype)
```

Sr.N o.	Parameter & Description
1	n The number of rows in the resulting matrix
2	M The number of columns, defaults to n
3	k Index of diagonal
4	dtype Data type of the output

Example

```
import numpy.matlib  
import numpy as np  
print np.matlib.eye(n = 3, M = 4, k = 0, dtype = float)
```

It will produce the following output –

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

[numpy.matlib.identity\(\)](#)

The **numpy.matlib.identity()** function returns the Identity matrix of the given size. An identity matrix is a square matrix with all diagonal elements as 1.

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

It will produce the following 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.]]
```

[numpy.matlib.rand\(\)](#)

The **numpy.matlib.rand()** function returns a matrix of the given size filled with random values.

Example

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

It will produce the following output –

```
[[ 0.82674464  0.57206837  0.15497519]
 [ 0.33857374  0.35742401  0.90895076]
 [ 0.03968467  0.13962089  0.39665201]]
```

Note : that a matrix is always two-dimensional, whereas ndarray is an n-dimensional array.

NumPy - Linear Algebra

NumPy package contains **numpy.linalg** module that provides all the functionality required for linear algebra.

[numpy.dot\(\)](#)

This function returns the dot product of two arrays. For 2-D vectors, it is the equivalent to matrix multiplication. For 1-D arrays, it is the inner product of the vectors. For N-dimensional arrays, it is a sum product over the **last axis of a** and the **second-last axis of b**.

```
import numpy.matlib
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])
np.dot(a,b)
```

```
[[37 40]
```

```
[85 92]]
```

Note that the dot product is calculated as –

```
[[1*11+2*13, 1*12+2*14], [3*11+4*13, 3*12+4*14]]
```


[numpy.vdot\(\)](#)

This function returns the dot product of the two vectors. If the argument is multi-dimensional array, it is flattened.

```
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])
print np.vdot(a,b)
```

It will produce the following output –

130

Note – $1*11 + 2*12 + 3*13 + 4*14 = 130$

numpy.inner()

This function returns the inner product of vectors for 1-D arrays. For higher dimensions, it returns the sum product over the last axes.

```
import numpy as np
print np.inner(np.array([1,2,3]), np.array([0,1,0]))
# Equates to 1*0+2*1+3*0
```

It will produce the following output –

2

```
# Multi-dimensional array example
import numpy as np
a = np.array([[1,2], [3,4]])
print 'Array a:'
print a

b = np.array([[11, 12], [13, 14]])
print 'Array b:'
print b

print 'Inner product:'
print np.inner(a,b)
```

```
Array a:
[[1 2]
 [3 4]]
Array b:
[[11 12]
 [13 14]]
Inner product:
[[35 41]
 [81 95]]
```

the inner product is calculated as –

```
1*11+2*12, 1*13+2*14
3*11+4*12, 3*13+4*14
```

[numpy.matmul\(\)](#)


- The **numpy.matmul()** function returns the matrix product of two arrays. While it returns a normal product for 2-D arrays, if dimensions of either argument is >2, it is treated as a stack of matrices residing in the last two indexes and is broadcast accordingly.
- On the other hand, if either argument is 1-D array, it is promoted to a matrix by appending a 1 to its dimension, which is removed after multiplication.

For 2-D array, it is matrix multiplication

```
import numpy.matlib
import numpy as np
a = [[1,0],[0,1]]
b = [[4,1],[2,2]]
print np.matmul(a,b)
```

It will produce the following output –

```
[[4 1]
 [2 2]]
```



```
# 2-D mixed with 1-D
import numpy.matlib
import numpy as np
a = [[1,0],[0,1]]
b = [1,2]
print np.matmul(a,b)
print np.matmul(b,a)
```

It will produce the following output –

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

```
# one array having dimensions > 2
import numpy.matlib
import numpy as np
a = np.arange(8).reshape(2,2,2)
b = np.arange(4).reshape(2,2)
print np.matmul(a,b)
```

It will produce the following output –

```
[[[2 3]
  [6 11]]
 [[10 19]
  [14 27]]]
```

NumPy - Determinant

- Determinant is a very useful value in linear algebra. It is calculated from the diagonal elements of a square matrix. For a 2x2 matrix, it is simply the subtraction of the product of the top left and bottom right element from the product of the other two.
- In other words, for a matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the determinant is computed as $ad - bc$. The larger square matrices are considered to be a combination of 2x2 matrices.
- The **numpy.linalg.det()** function calculates the determinant of the input matrix.

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

It will produce the following output –

-2.0

```
import numpy as np
b = np.array([[6,1,1], [4, -2, 5], [2,8,7]])
print b
print np.linalg.det(b)
print 6*(-2*7 - 5*8) - 1*(4*7 - 5*2) + 1*(4*8 - -2*2)
```

It will produce the following output –

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

-306.0

-306

$$\begin{array}{ccc} 6 & 1 & 1 \\ 4 & -2 & 5 \\ 2 & 8 & 7 \end{array}$$
$$6*(-2*7 - 5*8) - 1*(4*7 - 5*2) + 1(4*8 - -2*2)$$