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.

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

Operations using NumPy

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 – A Replacement for MatLab

NumPy is often used along with packages like **SciPy** (Scientific Python) and **Mat−plotlib** (plotting library). This combination is widely used as a replacement for MatLab, a popular platform for technical computing. However, Python alternative to MatLab is now seen as a more modern and complete programming language.

It is open source, which is an added advantage of NumPy.

Environment Setup

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>python –m pip install numpy
```

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

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.

alias: In Python alias are an alternate name 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)
```

Checking NumPy Version

The version string is stored under `__version__` attribute.

Example

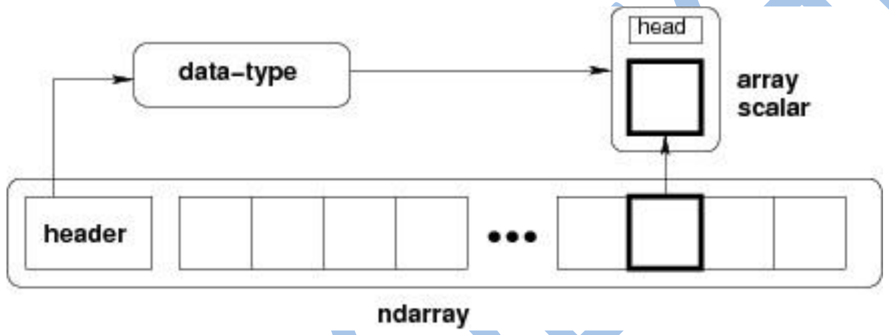
```
import numpy as np

print(np.__version__)
```

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**).

Any item extracted from ndarray object (by slicing) is represented by a Python object of one of array scalar types. The following diagram shows a relationship between ndarray, data type object (dtype) and array scalar type −



An instance of ndarray class can be constructed by different array creation routines described later in the tutorial. The basic ndarray is created using an array function in NumPy as follows −

```
numpy.array
```

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

```
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	**object** Any object exposing the array interface method returns an array, or any (nested) sequence.
2	**dtype** Desired data type of array, optional
3	**copy**

	Optional. By default (true), the object is copied
4	**order** C (row major) or F (column major) or A (any) (default)
5	**subok** By default, returned array forced to be a base class array. If true, sub-classes passed through
6	**ndmin** Specifies minimum dimensions of resultant array

Take a look at the following examples to understand better.

Example 1

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

The output is as follows −

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

Example 2

```
# more than one dimensions
import numpy as np
a = np.array([[1, 2], [3, 4]])
print a
```

The output is as follows −

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

Example 3

```
# minimum dimensions
import numpy as np
a = np.array([1, 2, 3,4,5], ndmin = 2)
print a
```

The output is as follows −

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

Example 4

```
# dtype parameter
import numpy as np
a = np.array([1, 2, 3], dtype = complex)
print a
```

The output is as follows −

```
[ 1.+0.j,  2.+0.j,  3.+0.j]
```

Example 5

```
import numpy as np
# Creating array from list with type float
a = np.array([[1, 2, 4], [5, 8, 7]], dtype = 'float')
print ("Array created using passed list:\n", a)

# Creating array from tuple
b = np.array((1 , 3, 2))
print ("\nArray created using passed tuple:\n", b)

# Creating a 3X4 array with all zeros
c = np.zeros((3, 4))
print ("\nAn array initialized with all zeros:\n", c)

# Create a constant value array of complex type
d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s."
            "Array type is complex:\n", d)
```

The output is as follows −

```
Array created using passed list:
 [[1. 2. 4.]
 [5. 8. 7.]]

Array created using passed tuple:
 [1 3 2]

An array initialized with all zeros:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

An array initialized with all 6s.Array type is complex:
 [[6.+0.j 6.+0.j 6.+0.j]
 [6.+0.j 6.+0.j 6.+0.j]
 [6.+0.j 6.+0.j 6.+0.j]]
```

The **ndarray** object consists of contiguous one-dimensional segment of computer memory, combined with an indexing scheme that maps each item to a location in the memory block. The memory block holds the elements in a row-major order (C style) or a column-major order (FORTRAN or MatLab style).

NumPy Data Types

NumPy supports a much greater variety of numerical types than Python does. The following table shows different scalar data types defined in NumPy.

Sr.No.	Data Types & Description
1	**bool_** Boolean (True or False) stored as a byte
2	**int_** Default integer type (same as C long; normally either int64 or int32)
3	**intc** Identical to C int (normally int32 or int64)
4	**intp** Integer used for indexing (same as C ssize_t; normally either int32 or int64)
5	**int8** Byte (-128 to 127)
6	**int16** Integer (-32768 to 32767)
7	**int32** Integer (-2147483648 to 2147483647)
8	**int64** Integer (-9223372036854775808 to 9223372036854775807)
9	**uint8** Unsigned integer (0 to 255)
10	**uint16** Unsigned integer (0 to 65535)
11	**uint32** Unsigned integer (0 to 4294967295)

12	**uint64** Unsigned integer (0 to 18446744073709551615)
13	**float_** Shorthand for float64
14	**float16** Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
15	**float32** Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
16	**float64** Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
17	**complex_** Shorthand for complex128
18	**complex64** Complex number, represented by two 32-bit floats (real and imaginary components)
19	**complex128** Complex number, represented by two 64-bit floats (real and imaginary components)

NumPy numerical types are instances of dtype (data-type) objects, each having unique characteristics. The dtypes are available as np.bool_, np.float32, etc.

Data Type Objects (dtype)

A data type object describes interpretation of fixed block of memory corresponding to an array, depending on the following aspects −

- Type of data (integer, float or Python object)
- Size of data
- Byte order (little-endian or big-endian)
- In case of structured type, the names of fields, data type of each field and part of the memory block taken by each field.
- If data type is a subarray, its shape and data type

The byte order is decided by prefixing '<' or '>' to data type. '<' means that encoding is little-endian (least significant is stored in smallest address). '>' means that encoding is big-endian (most significant byte is stored in smallest address).

A dtype object is constructed using the following syntax −

```
numpy.dtype(object, align, copy)
```

The parameters are −

- **Object** − To be converted to data type object
- **Align** − If true, adds padding to the field to make it similar to C-struct
- **Copy** − Makes a new copy of dtype object. If false, the result is reference to builtin data type object

Example 1

```
# using array-scalar type
import numpy as np
dt = np.dtype(np.int32)
print dt
```

The output is as follows −

```
int32
```

Example 2

```
#int8, int16, int32, int64 can be replaced by equivalent string 'i1',
'i2','i4', etc.
import numpy as np

dt = np.dtype('i4')
print dt
```

The output is as follows −

```
int32
```

Example 3

```
# using endian notation
import numpy as np
dt = np.dtype('>i4')
print dt
```

The output is as follows −

```
>i4
```

The following examples show the use of structured data type. Here, the field name and the corresponding scalar data type are to be declared.

Example 4

```
# first create structured data type
import numpy as np
dt = np.dtype([('age',np.int8)])
print dt
```

The output is as follows −

```
[('age', 'i1')]
```

Example 5

```
# now apply it to ndarray object
import numpy as np

dt = np.dtype([('age',np.int8)])
a = np.array([(10,),(20,),(30,)], dtype = dt)
print a
```

The output is as follows −

```
[(10,) (20,) (30,)]
```

Example 6

```
# file name can be used to access content of age column
import numpy as np

dt = np.dtype([('age',np.int8)])
a = np.array([(10,),(20,),(30,)], dtype = dt)
print a['age']
```

The output is as follows −

```
[10 20 30]
```

Example 7

The following examples define a structured data type called **student** with a string field 'name', an **integer field** 'age' and a **float field** 'marks'. This dtype is applied to ndarray object.

```
import numpy as np
student = np.dtype([('name','S20'), ('age', 'i1'), ('marks', 'f4')])
print student
```

The output is as follows −

```
[('name', 'S20'), ('age', 'i1'), ('marks', '<f4')])
```

Example 8

```
import numpy as np

student = np.dtype([('name','S20'), ('age', 'i1'), ('marks', 'f4')])
a = np.array([('abc', 21, 50),('xyz', 18, 75)], dtype = student)
print a
```

The output is as follows −

```
[('abc', 21, 50.0), ('xyz', 18, 75.0)]
```

Example 9

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

The output is as follows −

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

Each built-in data type has a character code that uniquely identifies it.

- **'b'** − boolean
- **'i'** − (signed) integer
- **'u'** − unsigned integer
- **'f'** − floating-point
- **'c'** − complex-floating point
- **'m'** − timedelta
- **'M'** − datetime
- **'O'** − (Python) objects
- **'S', 'a'** − (byte-)string
- **'U'** − Unicode
- **'V'** − raw data (void)

NumPyArray Creation

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))
```

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 `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).

nested array: are arrays that have arrays as their elements.

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

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.

NumPy has a whole sub module dedicated towards matrix operations called `numpy.mat`

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)
```

Check Number of Dimensions?

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

Example

Check how many dimensions the arrays have:

```
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.

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)
```

The constructor takes the following parameters.

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

The following examples show how you can use the **asarray** function.

Example 1

```
# convert list to ndarray
import numpy as np

x = [1,2,3]
a = np.asarray(x)
print a
```

Its output would be as follows −

```
[1  2  3]
```

Example 2

```
# dtype is set
import numpy as np

x = [1,2,3]
a = np.asarray(x, dtype = float)
print a
```

Now, the output would be as follows −

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

Example 3

```
# ndarray from tuple
import numpy as np

x = (1,2,3)
a = np.asarray(x)
print a
```

Its output would be −

```
[1  2  3]
```

Example 4

```
# ndarray from list of tuples
import numpy as np

x = [(1,2,3),(4,5)]
a = np.asarray(x)
print a
```

Here, the output would be as follows −

```
[(1, 2, 3) (4, 5)]
```

numpy.frombuffer

This function interprets a buffer as one-dimensional array. Any object that exposes the buffer interface is used as parameter to return an **ndarray**.

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

The constructor takes the following parameters.

Sr.No.	Parameter & Description
1	**buffer** Any object that exposes buffer interface
2	**dtype** Data type of returned ndarray. Defaults to float
3	**count** The number of items to read, default -1 means all data
4	**offset** The starting position to read from. Default is 0

Example

The following examples demonstrate the use of **frombuffer** function.

```
import numpy as np
s = 'Hello World'
a = np.frombuffer(s, dtype = 'S1')
print a
```

Here is its output −

```
['H'  'e'  'l'  'l'  'o'  ' '  'W'  'o'  'r'  'l'  'd']
```

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)
```

Here, the constructor takes the following parameters.

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 1

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

Its output is as follows −

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

Example 2

```
# 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
```

Now, the output would be as follows −

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

Arrays within the numerical range

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)
```

The constructor takes the following parameters.

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

The following examples show how you can use this function.

Example 1

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

Its output would be as follows −

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

Example 2

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

Here, the output would be −

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

Example 3

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

Its output is as follows −

```
[10  12  14  16  18]
```

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)
```

The constructor takes the following parameters.

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**

The following examples demonstrate the use **linspace** function.

Example 1

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

Its output would be −

```
[10.   12.5   15.   17.5  20.]
```

Example 2

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

The output would be −

```
[10.   12.   14.   16.   18.]
```

Example 3

```
# find retstep value
import numpy as np

x = np.linspace(1,2,5, retstep = True)
print x
# retstep here is 0.25
```

Now, the output would be −

```
(array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ]), 0.25)
```

numpy.logspace

This function returns an **ndarray** object that contains the numbers that are evenly spaced on a log scale. Start and stop endpoints of the scale are indices of the base, usually 10.

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

Following parameters determine the output of **logspace** function.

Sr.No.	Parameter & Description
1	**start**

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

The following examples will help you understand the **logspace** function.

Example 1

```
import numpy as np
# default base is 10
a = np.logspace(1.0, 2.0, num = 10)
print a
```

Its output would be as follows −

```
[ 10.           12.91549665     16.68100537      21.5443469  27.82559402
  35.93813664   46.41588834     59.94842503      77.42636827    100.    ]
```

Example 2

```
# set base of log space to 2
import numpy as np
a = np.logspace(1,10,num = 10, base = 2)
print a
```

Now, the output would be −

```
[ 2.     4.     8.    16.     32.     64.    128.    256.     512.    1024.]
```

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 1

```
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.

Broadcasting is possible if the following rules are satisfied −

- Array with smaller **ndim** than the other is prepended with '1' in its shape.
- Size in each dimension of the output shape is maximum of the input sizes in that dimension.
- An input can be used in calculation, if its size in a particular dimension matches the output size or its value is exactly 1.
- If an input has a dimension size of 1, the first data entry in that dimension is used for all calculations along that dimension.

A set of arrays is said to be **broadcastable** if the above rules produce a valid result and one of the following is true −

- Arrays have exactly the same shape.
- Arrays have the same number of dimensions and the length of each dimension is either a common length or 1.
- Array having too few dimensions can have its shape prepended with a dimension of length 1, so that the above stated property is true.

The following program shows an example of broadcasting.

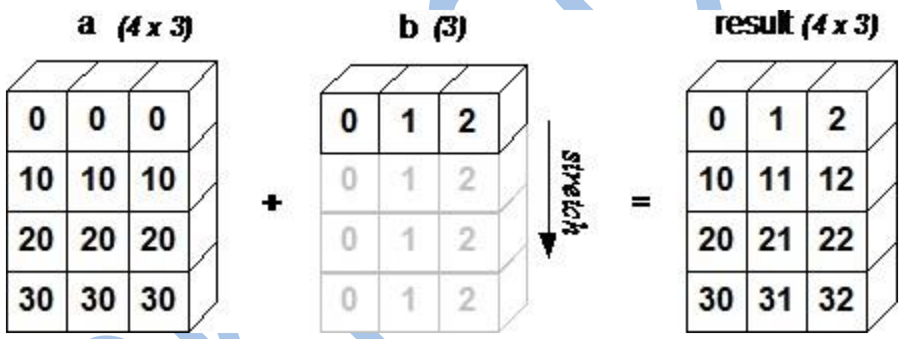
Example 2

```
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],[30.0,30.0,30.0]])
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.]]
```

The following figure demonstrates how array **b** is broadcast to become compatible with **a**.



NumPy Array Iteration

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.

Let us create a 3X4 array using arange() function and iterate over it using **nditer**.

Example 1

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
```

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

print 'Modified array is:'
for x in np.nditer(a):
   print x,
```

The output of this program is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Modified array is:
0 5 10 15 20 25 30 35 40 45 50 55
```

Example 2

The order of iteration is chosen to match the memory layout of an array, without considering a particular ordering. This can be seen by iterating over the transpose of the above array.

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)

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

print 'Transpose of the original array is:'
b = a.T
print b
print '\n'

print 'Modified array is:'
for x in np.nditer(b):
   print x,
```

The output of the above program is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Transpose of the original array is:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]

Modified array is:
```

```
0 5 10 15 20 25 30 35 40 45 50 55
```

Iteration Order

If the same elements are stored using F-style order, the iterator chooses the more efficient way of iterating over an array.

Example 1

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print 'Original array is:'
print a
print '\n'

print 'Transpose of the original array is:'
b = a.T
print b
print '\n'

print 'Sorted in C-style order:'
c = b.copy(order = 'C')
print c
for x in np.nditer(c):
   print x,

print '\n'

print 'Sorted in F-style order:'
c = b.copy(order = 'F')
print c
for x in np.nditer(c):
   print x,
```

Its output would be as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Transpose of the original array is:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]

Sorted in C-style order:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]
0 20 40 5 25 45 10 30 50 15 35 55
```

```
Sorted in F-style order:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]
0 5 10 15 20 25 30 35 40 45 50 55
```

Example 2

It is possible to force **nditer** object to use a specific order by explicitly mentioning it.

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)

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

print 'Sorted in C-style order:'
for x in np.nditer(a, order = 'C'):
   print x,
print '\n'

print 'Sorted in F-style order:'
for x in np.nditer(a, order = 'F'):
   print x,
```

Its output would be −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Sorted in C-style order:
0 5 10 15 20 25 30 35 40 45 50 55

Sorted in F-style order:
0 20 40 5 25 45 10 30 50 15 35 55
```

Modifying Array Values

The **nditer** object has another optional parameter called **op_flags**. Its default value is read-only, but can be set to read-write or write-only mode. This will enable modifying array elements using this iterator.

Example

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print 'Original array is:'
print a
print '\n'

for x in np.nditer(a, op_flags = ['readwrite']):
   x[...] = 2*x
print 'Modified array is:'
print a
```

Its output is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Modified array is:
[[ 0 10 20 30]
 [ 40 50 60 70]
 [ 80 90 100 110]]
```

External Loop

The nditer class constructor has a **'flags'** parameter, which can take the following values −

Sr.No.	Parameter & Description
1	**c_index** C_order index can be tracked
2	**f_index** Fortran_order index is tracked
3	**multi-index** Type of indexes with one per iteration can be tracked
4	**external_loop** Causes values given to be one-dimensional arrays with multiple values instead of zero-dimensional array

Example

In the following example, one-dimensional arrays corresponding to each column is traversed by the iterator.

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)

print 'Original array is:'
print a
print '\n'
print 'Modified array is:'
for x in np.nditer(a, flags = ['external_loop'], order = 'F'):
   print x,
```

The output is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Modified array is:
[ 0 20 40] [ 5 25 45] [10 30 50] [15 35 55]
```

Broadcasting Iteration

If two arrays are **broadcastable**, a combined **nditer** object is able to iterate upon them concurrently. Assuming that an array **a** has dimension 3X4, and there is another array **b** of dimension 1X4, the iterator of following type is used (array **b** is broadcast to size of **a**).

Example

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)

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

print 'Second array is:'
b = np.array([1, 2, 3, 4], dtype = int)
print b
print '\n'

print 'Modified array is:'
for x,y in np.nditer([a,b]):
   print "%d:%d" % (x,y),
```

Its output would be as follows −

```
First array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Second array is:
[1 2 3 4]

Modified array is:
0:1 5:2 10:3 15:4 20:1 25:2 30:3 35:4 40:1 45:2 50:3 55:4
```

NumPy Bitwise Operators

Following are the functions for bitwise operations available in NumPy package.

Sr.No.	Operation & Description
1	bitwise_and Computes bitwise AND operation of array elements
2	bitwise_or Computes bitwise OR operation of array elements
3	invert Computes bitwise NOT
4	left_shift Shifts bits of a binary representation to the left
5	right_shift Shifts bits of binary representation to the right

Bitwise and

The bitwise AND operation on the corresponding bits of binary representations of integers in input arrays is computed by np.bitwise_and() function.

Example

```
import numpy as np
print 'Binary equivalents of 13 and 17:'
a,b = 13,17
print bin(a), bin(b)
print '\n'

print 'Bitwise AND of 13 and 17:'
print np.bitwise_and(13, 17)
```

Its output is as follows −

```
Binary equivalents of 13 and 17:
0b1101 0b10001

Bitwise AND of 13 and 17: 1
```

You can verify the output as follows. Consider the following bitwise AND truth table.

A	B	AND
1	1	1
1	0	0
0	1	0
0	0	0

Bitwise or

The bitwise OR operation on the corresponding bits of binary representations of integers in input arrays is computed by **np.bitwise_or()** function.

Example

```
import numpy as np
a,b = 13,17
print 'Binary equivalents of 13 and 17:'
print bin(a), bin(b)

print 'Bitwise OR of 13 and 17:'
print np.bitwise_or(13, 17)
```

Its output is as follows −

```
Binary equivalents of 13 and 17:
0b1101 0b10001

Bitwise OR of 13 and 17:
29
```

You can verify this output using the following table. Consider the following Bitwise OR truth table.

A	B	OR
1	1	1
1	0	1
0	1	1
0	0	0

Bitwise invert

This function computes the bitwise NOT result on integers in the input array. For signed integers, two's complement is returned.

Example

```
import numpy as np
```

```
print 'Invert of 13 where dtype of ndarray is uint8:'
print np.invert(np.array([13], dtype = np.uint8))
print '\n'
# Comparing binary representation of 13 and 242, we find the inversion of
bits

print 'Binary representation of 13:'
print np.binary_repr(13, width = 8)
print '\n'

print 'Binary representation of 242:'
print np.binary_repr(242, width = 8)
```

Its output is as follows −

```
Invert of 13 where dtype of ndarray is uint8:
[242]

Binary representation of 13:
00001101

Binary representation of 242:
11110010
```

Note that **np.binary_repr()** function returns the binary representation of the decimal number in the given width.

Bitwise left_shift

The **numpy.left_shift()** function shifts the bits in binary representation of an array element to the left by specified positions. Equal number of 0s are appended from the right.

For example,

```
import numpy as np

print 'Left shift of 10 by two positions:'
print np.left_shift(10,2)
print '\n'

print 'Binary representation of 10:'
print np.binary_repr(10, width = 8)
print '\n'

print 'Binary representation of 40:'
print np.binary_repr(40, width = 8)
# Two bits in '00001010' are shifted to left and two 0s appended from right.
```

Its output is as follows −

```
Left shift of 10 by two positions:
40
```

```
Binary representation of 10:
00001010

Binary representation of 40:
00101000
```

Bitwise right_shift

The **numpy.right_shift()** function shift the bits in the binary representation of an array element to the right by specified positions, and an equal number of 0s are appended from the left.

```
import numpy as np

print 'Right shift 40 by two positions:'
print np.right_shift(40,2)
print '\n'

print 'Binary representation of 40:'
print np.binary_repr(40, width = 8)
print '\n'

print 'Binary representation of 10'
print np.binary_repr(10, width = 8)
# Two bits in '00001010' are shifted to right and two 0s appended from left.
```

Its output would be as follows −

```
Right shift 40 by two positions:
10

Binary representation of 40:
00101000

Binary representation of 10
00001010
```

NumPy String Functions

The following functions are used to perform vectorized string operations for arrays of dtype numpy.string_ or numpy.unicode_. They are based on the standard string functions in Python's built-in library.

Sr.No.	Function & Description
1	add() Returns element-wise string concatenation for two arrays of str or Unicode This function performs element wise string concatenation.

```
import numpy as np
print 'Concatenate two strings:'
print np.char.add(['hello'],[' xyz'])
print '\n'

print 'Concatenation example:'
print np.char.add(['hello', 'hi'],[' abc', ' xyz'])
```

Its output would be as follows −

```
Concatenate two strings:
['hello xyz']

Concatenation example:
['hello abc' 'hi xyz']
```

2 multiply()

Returns the string with multiple concatenation, element-wise

This function performs multiple concatenation.

```
import numpy as np
print np.char.multiply('Hello ',3)
```

Its output would be as follows −

```
Hello Hello Hello
```

3 center()

Returns a copy of the given string with elements centered in a string of specified length

This function returns an array of the required width so that the input string is centered and padded on the left and right with **fillchar**.

```
import numpy as np
# np.char.center(arr, width,fillchar)
print np.char.center('hello', 20,fillchar = '*')
```

Here is its output −

```
*******hello********
```

4 capitalize()

Returns a copy of the string with only the first character capitalized

This function returns the copy of the string with the first letter capitalized.

```
import numpy as np
print np.char.capitalize('hello world')
```

<table>
<tr><td></td><td>Its output would be −
<pre>Hello world</pre></td></tr>
<tr><td>5</td><td>title()

Returns the element-wise title cased version of the string or unicode

This function returns a title cased version of the input string with the first letter of each word capitalized.
<pre>import numpy as np
print np.char.title('hello how are you?')</pre>Its output would be as follows −
<pre>Hello How Are You?</pre></td></tr>
<tr><td>6</td><td>lower()

Returns an array with the elements converted to lowercase

This function returns an array with elements converted to lowercase. It calls **str.lower** for each element.
<pre>import numpy as np
print np.char.lower(['HELLO','WORLD'])
print np.char.lower('HELLO')</pre>Its output is as follows −
<pre>['hello' 'world']
hello</pre></td></tr>
<tr><td>7</td><td>upper()

Returns an array with the elements converted to uppercase

This function calls **str.upper** function on each element in an array to return the uppercase array elements.
<pre>import numpy as np
print np.char.upper('hello')
print np.char.upper(['hello','world'])</pre>Here is its output −
<pre>HELLO
['HELLO' 'WORLD']</pre></td></tr>
<tr><td>8</td><td>split()</td></tr>
</table>

	Returns a list of the words in the string, using separator delimiter
	
	This function returns a list of words in the input string. By default, a whitespace is used as a separator. Otherwise the specified separator character is used to spilt the string.
	
	`import numpy as np`
	`print np.char.split ('hello how are you?')`
	`print np.char.split ('TutorialsPoint,Hyderabad,Telangana', sep = ',')`
	
	Its output would be –
	
	`['hello', 'how', 'are', 'you?']`
	`['TutorialsPoint', 'Hyderabad', 'Telangana']`

9	splitlines()
	
	Returns a list of the lines in the element, breaking at the line boundaries
	
	This function returns a list of elements in the array, breaking at line boundaries.
	
	`import numpy as np`
	`print np.char.splitlines('hello\nhow are you?')`
	`print np.char.splitlines('hello\rhow are you?')`
	
	Its output is as follows –
	
	`['hello', 'how are you?']`
	`['hello', 'how are you?']`
	
	'\n', '\r', '\r\n' can be used as line boundaries.

10	strip()
	
	Returns a copy with the leading and trailing characters removed
	
	This function returns a copy of array with elements stripped of the specified characters leading and/or trailing in it.
	
	`import numpy as np`
	`print np.char.strip('ashok arora','a')`
	`print np.char.strip(['arora','admin','java'],'a')`
	
	Here is its output –
	
	`shok aror`
	`['ror' 'dmin' 'jav']`

11	join() Returns a string which is the concatenation of the strings in the sequence

	This method returns a string in which the individual characters are joined by separator character specified.
	
	`import numpy as np`
	`print np.char.join(':','dmy')`
	`print np.char.join([':','-'],['dmy','ymd'])`
	
	Its output is as follows −
	
	`d:m:y`
	`['d:m:y' 'y-m-d']`

12	replace()
	
	Returns a copy of the string with all occurrences of substring replaced by the new string
	
	This function returns a new copy of the input string in which all occurrences of the sequence of characters is replaced by another given sequence.
	
	`import numpy as np`
	`print np.char.replace ('He is a good boy', 'is', 'was')`
	
	Its output is as follows −
	
	`He was a good boy`

13	decode()
	
	Calls str.decode element-wise
	
	This function calls **numpy.char.decode()** decodes the given string using the specified codec.
	
	`import numpy as np`
	
	`a = np.char.encode('hello', 'cp500')`
	`print a`
	`print np.char.decode(a,'cp500')`
	
	Its output is as follows −
	
	b'\x88\x85\x93\x93\x96'
	hello

14	encode()
	
	Calls str.encode element-wise
	
	This function calls **str.encode function** for each element in the array. Default encoding is utf_8, codecs available in standard Python library may be used.
	
	`import numpy as np`

	`a = np.char.encode('hello', 'cp500')`
	`print a`
	
	Its output is as follows –
	
	b'\x88\x85\x93\x93\x96'

These functions are defined in character array class (numpy.char). The older Numarray package contained chararray class. The above functions in numpy.char class are useful in performing vectorized string operations.

NumPy Mathematical Functions

Quite understandably, NumPy contains a large number of various mathematical operations. 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
print np.sin(a*np.pi/180)
print '\n'

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

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

Here is its output –

```
Sine of different angles:
[ 0.          0.5         0.70710678  0.8660254   1.        ]

Cosine values for angles in array:
[  1.00000000e+00   8.66025404e-01   7.07106781e-01   5.00000000e-01
   6.12323400e-17]

Tangent values for given angles:
[  0.00000000e+00   5.77350269e-01   1.00000000e+00   1.73205081e+00
```

```
   1.63312394e+16]
```

arcsin, arcos, 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.

Example

```
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)
print '\n'

print 'arccos and arctan functions behave similarly:'
cos = np.cos(a*np.pi/180)
print cos
print '\n'

print 'Inverse of cos:'
inv = np.arccos(cos)
print inv
print '\n'

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

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

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

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

Its output is as follows −

```
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:
[  1.00000000e+00   8.66025404e-01   7.07106781e-01   5.00000000e-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:
[  0.00000000e+00   5.77350269e-01   1.00000000e+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)
```

Where,

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)
```

It produces the following output −

```
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** <= **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)
```

It produces the following output −

```
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)
```

It will produce the following output −

```
The given array:
[ -1.7   1.5  -0.2   0.6  10. ]

The modified array:
[ -1.   2.  -0.   1.  10.]
```

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.

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)
```

It will produce the following 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]
```

numpy.ptp()

The **numpy.ptp()** function returns the range (maximum-minimum) of values along an axis.

```
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)
```

It will produce the following 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]
```

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. The function **numpy.percentile()** takes the following arguments.

```
numpy.percentile(a, q, axis)
```

Where,

Sr.No.	Argument & Description
1	**a** Input array
2	**q** The percentile to compute must be between 0-100
3	**axis** 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)
```

It will produce the following 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.]
```

numpy.median()

Median is defined as the value separating the higher half of a data sample from the lower half. The **numpy.median()** function is used as shown in the following program.

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)
```

It will produce the following 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.]
```

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.

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)
```

It will produce the following output −

```
Our array is:
[[1 2 3]
 [3 4 5]
 [4 5 6]]

Applying mean() function:
3.66666666667

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)

Example

```
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)
```

It will produce the following 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

```
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)
```

It will produce the following 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.]))
```

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
```

Sorting & Searching

A variety of sorting related functions are available in NumPy. These sorting functions implement different sorting algorithms, each of them characterized by the speed of execution, worst case performance, the workspace required and the stability of algorithms.

numpy.sort()

The sort() function returns a sorted copy of the input array. It has the following parameters −

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

Where,

Sr.No.	Parameter & Description
1	**a** Array to be sorted
2	**axis** The axis along which the array is to be sorted. If none, the array is flattened, sorting on the last axis
3	**kind** Default is quicksort
4	**order** If the array contains fields, the order of fields to be sorted

Example

```
import numpy as np
a = np.array([[3,7],[9,1]])

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

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

print 'Sort along axis 0:'
print np.sort(a, axis = 0)
print '\n'

# Order parameter in sort function
dt = np.dtype([('name', 'S10'),('age', int)])
a = np.array([("raju",21),("anil",25),("ravi", 17), ("amar",27)], dtype = dt)
```

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

print 'Order by name:'
print np.sort(a, order = 'name')
```

It will produce the following output −

```
Our array is:
[[3 7]
 [9 1]]

Applying sort() function:
[[3 7]
 [1 9]]

Sort along axis 0:
[[3 1]
 [9 7]]

Our array is:
[('raju', 21) ('anil', 25) ('ravi', 17) ('amar', 27)]

Order by name:
[('amar', 27) ('anil', 25) ('raju', 21) ('ravi', 17)]
```

numpy.argsort()

The **numpy.argsort()** function performs an indirect sort on input array, along the given axis and using a specified kind of sort to return the array of indices of data. This indices array is used to construct the sorted array.

Example

```
import numpy as np
x = np.array([3, 1, 2])

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

print 'Applying argsort() to x:'
y = np.argsort(x)
print y
print '\n'

print 'Reconstruct original array in sorted order:'
print x[y]
print '\n'

print 'Reconstruct the original array using loop:'
for i in y:
   print x[i],
```

It will produce the following output −

```
Our array is:
[3 1 2]

Applying argsort() to x:
[1 2 0]

Reconstruct original array in sorted order:
[1 2 3]

Reconstruct the original array using loop:
1 2 3
```

numpy.lexsort()

function performs an indirect sort using a sequence of keys. The keys can be seen as a column in a spreadsheet. The function returns an array of indices, using which the sorted data can be obtained. Note, that the last key happens to be the primary key of sort.

Example

```
import numpy as np

nm = ('raju','anil','ravi','amar')
dv = ('f.y.', 's.y.', 's.y.', 'f.y.')
ind = np.lexsort((dv,nm))

print 'Applying lexsort() function:'
print ind
print '\n'

print 'Use this index to get sorted data:'
print [nm[i] + ", " + dv[i] for i in ind]
```

It will produce the following output −

```
Applying lexsort() function:
[3 1 0 2]

Use this index to get sorted data:
['amar, f.y.', 'anil, s.y.', 'raju, f.y.', 'ravi, s.y.']
```

NumPy module has a number of functions for searching inside an array. Functions for finding the maximum, the minimum as well as the elements satisfying a given condition are available.

numpy.argmax() and numpy.argmin()

These two functions return the indices of maximum and minimum elements respectively along the given axis.

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 argmax() function:'
print np.argmax(a)
print '\n'

print 'Index of maximum number in flattened array'
print a.flatten()
print '\n'

print 'Array containing indices of maximum along axis 0:'
maxindex = np.argmax(a, axis = 0)
print maxindex
print '\n'

print 'Array containing indices of maximum along axis 1:'
maxindex = np.argmax(a, axis = 1)
print maxindex
print '\n'

print 'Applying argmin() function:'
minindex = np.argmin(a)
print minindex
print '\n'

print 'Flattened array:'
print a.flatten()[minindex]
print '\n'

print 'Flattened array along axis 0:'
minindex = np.argmin(a, axis = 0)
print minindex
print '\n'

print 'Flattened array along axis 1:'
minindex = np.argmin(a, axis = 1)
print minindex
```

It will produce the following output –

```
Our array is:
[[30 40 70]
 [80 20 10]
 [50 90 60]]

Applying argmax() function:
7

Index of maximum number in flattened array
[30 40 70 80 20 10 50 90 60]
```

```
Array containing indices of maximum along axis 0:
[1 2 0]

Array containing indices of maximum along axis 1:
[2 0 1]

Applying argmin() function:
5

Flattened array:
10

Flattened array along axis 0:
[0 1 1]

Flattened array along axis 1:
[0 2 0]
```

numpy.nonzero()

The **numpy.nonzero()** function returns the indices of non-zero elements in the input array.

Example

```
import numpy as np
a = np.array([[30,40,0],[0,20,10],[50,0,60]])

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

print 'Applying nonzero() function:'
print np.nonzero (a)
```

It will produce the following output −

```
Our array is:
[[30 40  0]
 [ 0 20 10]
 [50  0 60]]

Applying nonzero() function:
(array([0, 0, 1, 1, 2, 2]), array([0, 1, 1, 2, 0, 2]))
```

numpy.where()

The where() function returns the indices of elements in an input array where the given condition is satisfied.

Example

```
import numpy as np
x = np.arange(9.).reshape(3, 3)
```

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

print 'Indices of elements > 3'
y = np.where(x > 3)
print y

print 'Use these indices to get elements satisfying the condition'
print x[y]
```

It will produce the following output −

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

Indices of elements > 3
(array([1, 1, 2, 2, 2]), array([1, 2, 0, 1, 2]))

Use these indices to get elements satisfying the condition
[ 4.  5.  6.  7.  8.]
```

numpy.extract()

The **extract()** function returns the elements satisfying any condition.

```
import numpy as np
x = np.arange(9.).reshape(3,  3)

print 'Our array is:'
print x

# define a condition
condition = np.mod(x,2)  ==  0

print 'Element-wise value of condition'
print condition

print 'Extract elements using condition'
print np.extract(condition, x)
```

It will produce the following output −

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

Element-wise value of condition
[[ True False True]
 [False True False]
 [ True False True]]
```

```
Extract elements using condition
[ 0. 2. 4. 6. 8.]
```

Copies and Views

While executing the functions, some of them return a copy of the input array, while some return the view. When the contents are physically stored in another location, it is called **Copy**. If on the other hand, a different view of the same memory content is provided, we call it as **View**.

No Copy

Simple assignments do not make the copy of array object. Instead, it uses the same id() of the original array to access it. The **id()** returns a universal identifier of Python object, similar to the pointer in C.

Furthermore, any changes in either gets reflected in the other. For example, the changing shape of one will change the shape of the other too.

Example

```
import numpy as np
a = np.arange(6)

print 'Our array is:'
print a

print 'Applying id() function:'
print id(a)

print 'a is assigned to b:'
b = a
print b

print 'b has same id():'
print id(b)

print 'Change shape of b:'
b.shape = 3,2
print b

print 'Shape of a also gets changed:'
print a
```

It will produce the following output −

```
Our array is:
[0 1 2 3 4 5]

Applying id() function:
139747815479536
```

```
a is assigned to b:
[0 1 2 3 4 5]
b has same id():
139747815479536

Change shape of b:
[[0 1]
 [2 3]
 [4 5]]

Shape of a also gets changed:
[[0 1]
 [2 3]
 [4 5]]
```

View or Shallow Copy

NumPy has **ndarray.view()** method which is a new array object that looks at the same data of the original array. Unlike the earlier case, change in dimensions of the new array doesn't change dimensions of the original.

Example

```
import numpy as np
# To begin with, a is 3X2 array
a = np.arange(6).reshape(3,2)

print 'Array a:'
print a

print 'Create view of a:'
b = a.view()
print b

print 'id() for both the arrays are different:'
print 'id() of a:'
print id(a)
print 'id() of b:'
print id(b)

# Change the shape of b. It does not change the shape of a
b.shape = 2,3

print 'Shape of b:'
print b

print 'Shape of a:'
print a
```

It will produce the following output −

```
Array a:
[[0 1]
 [2 3]
 [4 5]]
```

```
Create view of a:
[[0 1]
 [2 3]
 [4 5]]

id() for both the arrays are different:
id() of a:
140424307227264
id() of b:
140424151696288

Shape of b:
[[0 1 2]
 [3 4 5]]

Shape of a:
[[0 1]
 [2 3]
 [4 5]]
```

Slice of an array creates a view.

Example

```
import numpy as np
a = np.array([[10,10], [2,3], [4,5]])

print 'Our array is:'
print a

print 'Create a slice:'
s = a[:, :2]
print s
```

It will produce the following output −

```
Our array is:
[[10 10]
 [ 2 3]
 [ 4 5]]

Create a slice:
[[10 10]
 [ 2 3]
 [ 4 5]]
```

Deep Copy

The **ndarray.copy()** function creates a deep copy. It is a complete copy of the array and its data, and doesn’t share with the original array.

Example

```
import numpy as np
```

```
a = np.array([[10,10], [2,3], [4,5]])

print 'Array a is:'
print a

print 'Create a deep copy of a:'
b = a.copy()
print 'Array b is:'
print b

#b does not share any memory of a
print 'Can we write b is a'
print b is a

print 'Change the contents of b:'
b[0,0] = 100

print 'Modified array b:'
print b

print 'a remains unchanged:'
print a
```

It will produce the following output –

```
Array a is:
[[10 10]
 [ 2 3]
 [ 4 5]]

Create a deep copy of a:
Array b is:
[[10 10]
 [ 2 3]
 [ 4 5]]
Can we write b is a
False

Change the contents of b:
Modified array b:
[[100 10]
 [ 2 3]
 [ 4 5]]

a remains unchanged:
[[10 10]
 [ 2 3]
 [ 4 5]]
```

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)
```

Where,

Sr.No.	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
```

It will produce the following 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)
```

Where,

Sr.No.	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. Both the objects are inter-convertible.

Example

```
import numpy.matlib
import numpy as np

i = np.matrix('1,2;3,4')
print i
```

It will produce the following output −

```
[[1  2]
 [3  4]]
```

Example

```
import numpy.matlib
import numpy as np

j = np.asarray(i)
print j
```

It will produce the following output −

```
[[1  2]
 [3  4]]
```

Example

```
import numpy.matlib
import numpy as np

k = np.asmatrix (j)
print k
```

It will produce the following output −

```
[[1  2]
 [3  4]]
```

NumPy Linear Algebra

NumPy package contains **numpy.linalg** module that provides all the functionality required for linear algebra. Some of the important functions in this module are described in the following table.

Sr.No.	Function & Description

1	dot
	
	Dot product of the two arrays
	
	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)`
	
	It will produce the following output −
	
	`[[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]]`

2	vdot
	
	Dot product of the two vectors
	
	This function returns the dot product of the two vectors. If the first argument is complex, then its conjugate is used for calculation. If the argument id is multi-dimensional array, it is flattened.
	
	Example
	
	`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

3	inner
	
	Inner product of the two arrays
	
	This function returns the inner product of vectors for 1-D arrays. For higher dimensions, it returns the sum product over the last axes.
	
	Example
	
	`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`
	
	Example
	
	`# 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)
```

It will produce the following output −

```
Array a:
[[1 2]
[3 4]]

Array b:
[[11 12]
[13 14]]

Inner product:
[[35 41]
[81 95]]
```

In the above case, the inner product is calculated as −

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

4 matmul

Matrix product of the two arrays

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.

Example

```
# 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]]
```

	Example
	
	`# 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]`
	
	Example
	
	`# 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]]]`

5	determinant
	
	Computes the determinant of the array
	
	Determinant is a very useful value in linear algebra. It 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 other two.
	
	In other words, for a matrix [[a,b], [c,d]], 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
```

Example

```
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
```

6 solve

Solves the linear matrix equation

The **numpy.linalg.solve()** function gives the solution of linear equations in the matrix form.

Considering the following linear equations −

x + y + z = 6

2y + 5z = -4

2x + 5y - z = 27

7 inv

Finds the multiplicative inverse of the matrix

We use **numpy.linalg.inv()** function to calculate the inverse of a matrix. The inverse of a matrix is such that if it is multiplied by the original matrix, it results in identity matrix.

Example

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]])
y = np.linalg.inv(x)
print x
print y
print np.dot(x,y)
```

It should produce the following output −

```
[[1 2]
 [3 4]]
[[-2.   1. ]
 [ 1.5 -0.5]]
[[  1.00000000e+00   1.11022302e-16]
 [  0.00000000e+00   1.00000000e+00]]
```

Example

Let us now create an inverse of matrix A in our example.

```
import numpy as np
a = np.array([[1,1,1],[0,2,5],[2,5,-1]])

print 'Array a:"
print a
ainv = np.linalg.inv(a)

print 'Inverse of a:'
print ainv

print 'Matrix B is:'
b = np.array([[6],[-4],[27]])
print b

print 'Compute A-1B:'
x = np.linalg.solve(a,b)
print x
# this is the solution to linear equations x = 5, y = 3, z = -2
```

It will produce the following output −

```
Array a:
[[ 1 1 1]
 [ 0 2 5]
 [ 2 5 -1]]

Inverse of a:
[[ 1.28571429 -0.28571429 -0.14285714]
 [-0.47619048 0.14285714 0.23809524]
 [ 0.19047619 0.14285714 -0.0952381 ]]

Matrix B is:
[[ 6]
```

	```
	 [-4]
	 [27]]

	Compute A-1B:
	[[ 5.]
	 [ 3.]
	 [-2.]]
	The same result can be obtained by using the function –
	x = np.dot(ainv,b)
	```