Introduction to NumPy

Jae Yun JUN KIM

February 21, 2018

1 NumPy environment

Standard Python distribution doesn't come bundled with NumPy module. A lightweight alternative is to install NumPy using popular Python package installer, pip.

```
pip install numpy
```

The best way to enable NumPy is to use an installable binary package specific to your operating system. These binaries contain full SciPy stack (inclusive of NumPy, SciPy, matplotlib, IPython, SymPy and nose packages along with core Python).

Windows
Anaconda (from https://www.continuum.io) is a free Python distribution for SciPy stack. It is also available for Linux and Mac.
Canopy (https://www.enthought.com/products/canopy/) is available as free as well as commercial distribution with full SciPy stack for Windows, Linux and Mac.
Python (x,y): It is a free Python distribution with SciPy stack and Spyder IDE for Windows OS. (Downloadable from http://python-xy.github.io/)

Linux
Package managers of respective Linux distributions are used to install one or more packages in SciPy stack.

Ubuntu

```
sudo apt-get install python-numpy python-scipy python-matplotlibipythonipython- notebook python-pandas python-sympy python-nose
```

To test whether NumPy module is properly installed, try to import it from Python prompt.

```
import numpy
```

Alternatively, NumPy package is imported using the following syntax:

```
import numpy as np
```

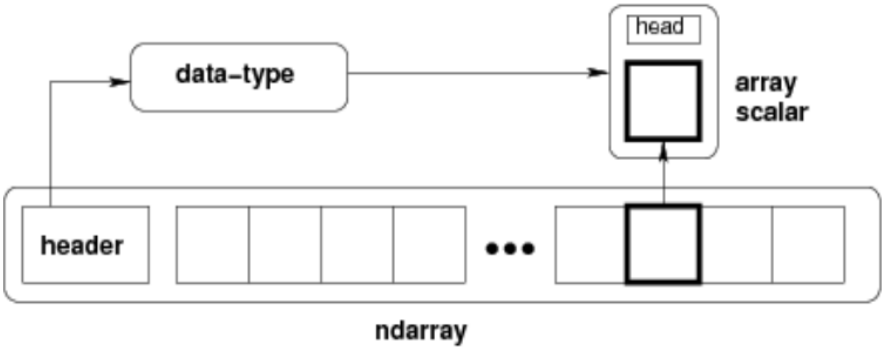
If it is not installed, the following error message will be displayed.

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import numpy
```

```
ImportError: No module named 'numpy'
```

2 *ndarray* Object

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:

object	Any object exposing the array interface method returns an array, or any (nested) sequence
dtype	Desired data type of array, optional
copy	Optional. By default (true), the object is copied
order	C (row major) or F (column major) or A (any) (default)
subok	By default, returned array forced to be a base class array. If true, sub-classes passed through
ndimin	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]
```

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).

3 Data type

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

Data Types	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

4 Array attributes

In this chapter, we will discuss the various array attributes of NumPy.

ndarray.shape

This array attribute returns a tuple consisting of array dimensions. It can also be used to resize the array.
Example 1

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

The output is as follows:

```
(2, 3)
```

Example 2

```
# this resizes the ndarray
import numpy as np
a=np.array([[1,2,3],[4,5,6]])
a.shape=(3,2)
print(a)
```

The output is as follows:

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

Example 3
NumPy also provides a reshape function to resize an array.

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

The output is as follows:

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

ndarray.ndim

This array attribute returns the number of array dimensions.
Example 1

```
# this is one dimensional array
import numpy as np
a = np.arange(24)
a.ndim

# now reshape it
b = a.reshape(2,4,3)
b.ndim
```

The output is as follows:

```
1


3
```

5 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.

ndarray.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.

Shape	Shape of an empty array in int or tuple of int
Dtype	Desired output data type. Optional
Order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example 1
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.

ndarray.zeros

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

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

The constructor takes the following parameters.

Shape	Shape of an empty array in int or sequence of int
Dtype	Desired output data type. Optional
Order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example 2

```
# 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 3

```
import numpy as np
x = np.zeros((5,), dtype=np.int)
print(x)
```

The output is as follows:

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

ndarray.ones

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

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

The constructor takes the following parameters.
Example 4

Shape	Shape of an empty array in int or sequence of int
Dtype	Desired output data type. Optional
Order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

```
# 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 5

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

The output is as follows:

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

6 Array from numerical ranges

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.

start	The start of an interval. If omitted, defaults to 0
stop	The end of an interval (not including this number)
step	Spacing between values, default is 1
dtype	Data type of resulting ndarray. If not given, data type of input is used

Example 1

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

The output is as follows:

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

Example 2

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

The output is as follows:

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

Example 3

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

The 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.
Example 4

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

The output is as follows:

start	The starting value of the sequence
stop	The end value of the sequence, included in the sequence if endpoint set to true
num	The number of evenly spaced samples to be generated. Default is 50
endpoint	True by default, hence the stop value is included in the sequence. If false, it is not included
retstep	If true, returns samples and step between the consecutive numbers
dtype	Data type of output **ndarray**

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

Example 5

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

The output is as follows:

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

Example 5

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

The output is as follows:

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

7 Indexing and slicing

Contents of ndarray object can be accessed and modified by indexing or slicing, just like Python's in-built container objects.
As mentioned earlier, items in ndarray object follows zero-based index. Three types of indexing methods are available: field access, basic slicing and advanced indexing.
Basic slicing is an extension of Python's basic concept of slicing to n dimensions. A Python slice object is constructed by giving start, stop, and step parameters to the built-in slice function. This slice object is passed to the array to extract a part of array.
Example 1

```
import numpy as np
a = np.arange(10)
s = slice(2,7,2)
print(a[s])
```

The output is as follows:

```
[2 4 6]
```

In the above example, an ndarray object is prepared by arange() function. Then a slice object is defined with start, stop, and step values 2, 7, and 2 respectively. When this slice object is passed to the ndarray, a part of it starting with index 2 up to 7 with a step of 2 is sliced.
The same result can also be obtained by giving the slicing parameters separated by a colon : (start:stop:step) directly to the ndarray object.
Example 2

```
import numpy as np
a = np.arange(10)
b = a[2:7:2]
print(b)
```

The output is as follows:

```
[2 4 6]
```

If only one parameter is put, a single item corresponding to the index will be returned. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index) with default step one are sliced.
Example 3

```
# slice single item
import numpy as np
a = np.arange(10)
b = a[5]
print(b)
```

The output is as follows:

```
5
```

Example 4

```
# slice items starting from index
import numpy as np
a = np.arange(10)
print(a[2:])
```

The output is as follows:

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

Example 5

```
# slice items between indexes
import numpy as np
a = np.arange(10)
print(a[2:5])
```

The output is as follows:

```
[2 3 4]
```

Example 6

```
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print(a)
# slice items starting from index
print('Now we will slice the array from the index a[1:]')
print(a[1:])
```

The output is as follows:

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

Now we will slice the array from the index a[1:]
[[3 4 5]
[4 5 6]]
```

Slicing can also include ellipsis (...) to make a selection tuple of the same length as the dimension of an array. If ellipsis is used at the row position, it will return an ndarray comprising of items in rows.
Example 7

```
# array to begin with
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print('Our array is:')
print(a)
print('\n')
# this returns array of items in the second column
print('The items in the second column are:')
print(a[...,1])
print('\n')
# Now we will slice all items from the second row
print('The items in the second row are:')
print(a[1,...])
print('\n')
```

```
# Now we will slice all items from column 1 onwards
print('The items column 1 onwards are:')
print(a[...,1:])
```

The output is as follows:

```
Our array is:
[[1 2 3]
[3 4 5]
[4 5 6]]
The items in the second column are:
[2 4 5]
The items in the second row are:
[3 4 5]
The items column 1 onwards are:
[[2 3]
[4 5]
[5 6]]
```

8 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

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))
```

The output is as follows:

```
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 1.00000000e+00
1.73205081e+00 6.12323400e-17]
```

```
Tangent values for given angles:
[0.00000000e+00 5.77350269e-01 1.63312394e+16]
```

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

a	Input data
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))
```

The output is as follows:

```
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))
```

The output is as follows:

```
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 output is as follows:

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

9 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))
```

The output is as follows:

```
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 1, 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))
```

The output is as follows:

```
Our array is:
[ 0.25 1.33 1. 0. 100. ]

After applying reciprocal function:

main.py:8: 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.
Example

```
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))
```

The output is as follows:

```
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.remainder() also produces the same result.
Example

```
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.remainder(a,b))
```

The output is as follows:

```
First array:
[10 20 30]

Second array:
[3 5 7]

Applying mod() function:
[1 0 2]

Applying remainder() function:
[1 0 2]
```

The following functions are used to perform operations on array with complex numbers.

- numpy.real() returns the real part of the complex data type argument.
- numpy.imag() returns the imaginary part of the complex data type argument.
- numpy.conj() returns the complex conjugate, which is obtained by changing the sign of the imaginary part.
- numpy.angle() returns the angle of the complex argument. The function has degree parameter. If true, the angle in the degree is returned, otherwise the angle is in radians.

Example

```
import numpy as np
a = np.array([-5.6j, 0.2j, 11. , 1+1j])
print('Our array is:')
print(a)
print('\n')
print('Applying real() function:')
print(np.real(a))
print('\n')
print('Applying imag() function:')
print(np.imag(a))
print('\n')
print('Applying conj() function:')
print(np.conj(a))
print('\n')
print('Applying angle() function:')
print(np.angle(a))
print('\n')
print('Applying angle() function again (result in degrees)')
print(np.angle(a, deg=True))
```

The output is as follows:

```
Our array is:
[ 0.-5.6j 0.+0.2j 11.+0.j 1.+1.j ]
Applying real() function:
[ 0. 0. 11. 1.]
Applying imag() function:
[-5.6 0.2 0. 1. ]
Applying conj() function:
[ 0.+5.6j 0.-0.2j 11.-0.j 1.-1.j ]
Applying angle() function:
[-1.57079633 1.57079633 0. 0.78539816]
Applying angle() function again (result in degrees)
[-90. 90. 0. 45.]
```

10 Matrix operations

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,

shape	**int** or tuple of **int** defining the shape of the new matrix
Dtype	Optional. Data type of the output
order	C or F

Example

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

The output is as follows:

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

matlib.zeros()

This function returns the matrix filled with zeros.
Example

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

The output is as follows:

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

matlib.ones()

This function returns the matrix filled with 1s.
Example

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

The output is as follows:

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

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,

n	The number of rows in the resulting matrix
M	The number of columns, defaults to n
k	Index of diagonal
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))
```

The output is as follows:

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

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.
Example

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

The output is as follows:

```
[[ 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.]]
```

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))
```

The output is as follows:

```
[[ 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)
```

The output is as follows:

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

Example

```
import numpy.matlib
import numpy as np
j = np.asarray(i)
print(j)
```

The output is as follows:

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

Example

```
import numpy.matlib
import numpy as np
k = np.asmatrix (j)
print(k)
```

The output is as follows:

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

11 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.

dot	Dot product of the two arrays
vdot	Dot product of the two vectors
inner	Inner product of the two arrays
matmul	Matrix product of the two arrays
det	Computes the determinant of the array
solve	Solves the linear matrix equation
inv	Finds the multiplicative inverse of the matrix

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.
Example

```
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)
```

The output is as follows:

```
[[37 40]
[85 92]]
```

numpy.vdot()

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))
```

The output is as follows:

```
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.
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
```

The output is as follows:

```
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))
```

The output is as follows:

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

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

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

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.
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))
```

The output is as follows:

```
[[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))
```

The output is as follows:

```
[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))
```

The output is as follows:

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

Determinant

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.
Example

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

The output is as follows:

```
-2.0
```

Example

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

The output is as follows:

```
-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))
```

The output is as follows:

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

-306.0

-306
```

numpy.linalg.solve()

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
```

They can be represented in the matrix form as:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -4 \\ 27 \end{bmatrix}$$

If these three matrices are called A, X and B, the equation becomes:

```
A*X = B
```

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))
```

The output is as follows:

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

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)^(-1) B:')
```

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

The output is as follows:

```
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)
```

12 Matplotlib

Matplotlib is a plotting library for Python. It is used along with NumPy to provide an environment that is an effective open source alternative for MatLab. It can also be used with graphics toolkits like PyQt and wxPython.
Matplotlib module was first written by John D. Hunter. Since 2012, Michael Droettboom is the principal developer. Currently, Matplotlib ver. 1.5.1 is the stable version available. The package is available in binary distribution as well as in the source code form on http://matplotlib.org.
Conventionally, the package is imported into the Python script by adding the following statement:

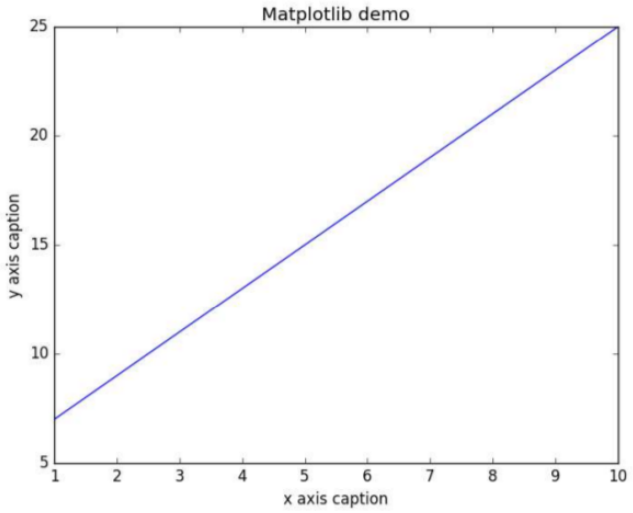
```
from matplotlib import pyplot as plt
```

Here pyplot() is the most important function in matplotlib library, which is used to plot 2D data. The following script plots the equation y=2x+5
Example

```
import numpy as np
from matplotlib import pyplot as plt
x=np.arange(1,11)
y=2*x+5
plt.title("Matplotlib demo")
plt.xlabel("x axis caption")
```

```
plt.ylabel("y axis caption")
plt.plot(x,y)
plt.show()
```

An ndarray object x is created from np.arange() function as the values on the x axis. The corresponding values on the y axis are stored in another ndarray object y. These values are plotted using plot() function of pyplot submodule of matplotlib package. The graphical representation is displayed by show() function.
The above code should produce the following output:



Instead of the linear graph, the values can be displayed discretely by adding a format string to the plot() function.
Example

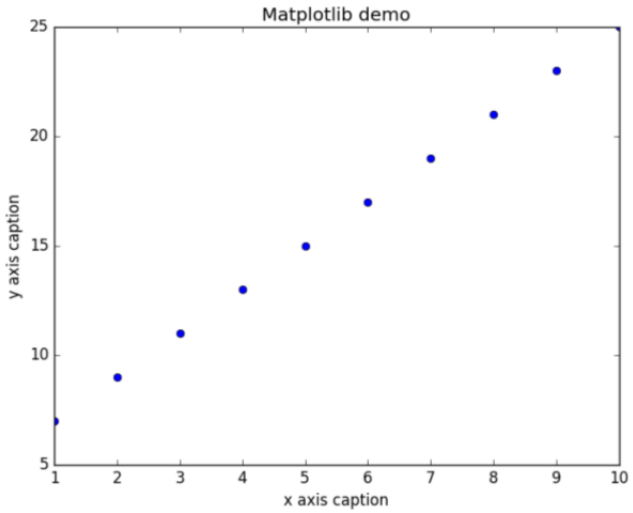
```
import numpy as np
from matplotlib import pyplot as plt
x=np.arange(1,11)
y=2*x+5
plt.title("Matplotlib demo")
plt.xlabel("x axis caption")
plt.ylabel("y axis caption")
plt.plot(x,y,"ob")
plt.show()
```

The above code should produce the following output:

Sine wave plot

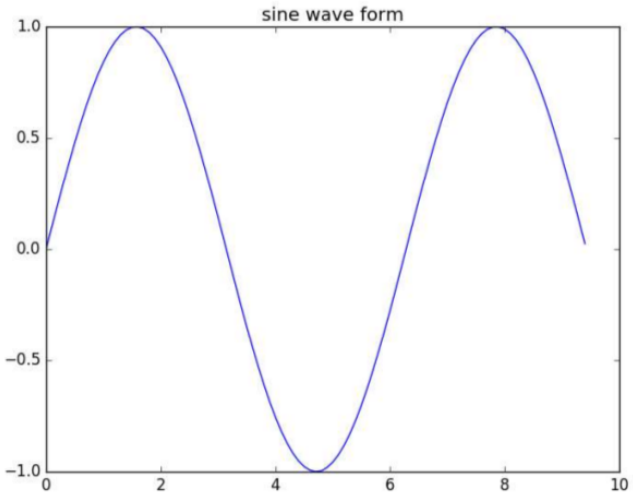
The following script produces the sine wave plot using matplotlib.
Example

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
```




```
plt.title("sine wave form")
# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```

This code should produce the following output:



subplot()

The subplot() function allows you to plot different things in the same figure. In the following script, sine and cosine values are plotted.
Example

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
```

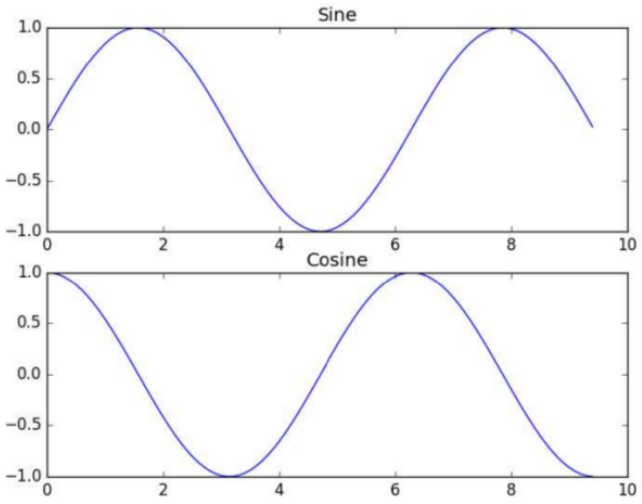
```
# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

This code should produce the following output:



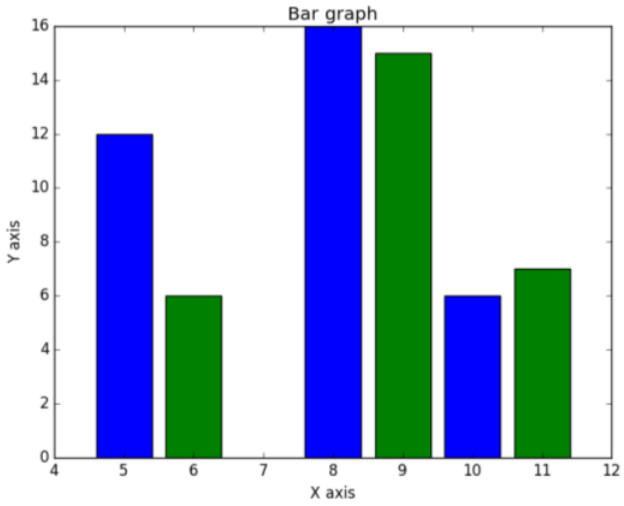
bar()

The pyplot submodule provides bar() function to generate bar graphs. The following example produces the bar graph of two sets of x and y arrays.
Example

```
from matplotlib import pyplot as plt
x =[5,8,10]
y =[12,16,6]
x2 =[6,9,11]
y2 =[6,15,7]
plt.bar(x, y, align='center')
plt.bar(x2, y2, color='g', align='center')
plt.title('Bar graph')
plt.ylabel('Y axis')
```

```
plt.xlabel('X axis')
plt.show()
```

This code should produce the following output:



13 I/O with NumPy

The ndarray objects can be saved to and loaded from the disk files. The IO functions available are:

- load() and save() functions handle /numPy binary files (with npy extension)
- loadtxt() and savetxt() functions handle normal text files

NumPy introduces a simple file format for ndarray objects. This .npy file stores data, shape, dtype and other information required to reconstruct the ndarray in a disk file such that the array is correctly retrieved even if the file is on another machine with different architecture.

numpy.save()

The numpy.save() file stores the input array in a disk file with npy extension.

```
import numpy as np
a=np.array([1,2,3,4,5])
np.save('outfile',a)
```

To reconstruct array from outfile.npy, use load() function.

```
import numpy as np
b = np.load('outfile.npy')
print(b)
```

It will produce the following output:

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

The save() and load() functions accept an additional Boolean parameter allow_pickles. A pickle in Python is used to serialize and de-serialize objects before saving to or reading from a disk file.

numpy.savetxt()

The storage and retrieval of array data in simple text file format is done with savetxt() and loadtxt() functions.

```
import numpy as np
a = np.array([1,2,3,4,5])
np.savetxt('out.txt',a)
b = np.loadtxt('out.txt')
print(b)
```

It will produce the following output:

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

The savetxt() and loadtxt() functions accept additional optional parameters such as header, footer, and delimiter.