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.

Operations using NumPy

Using NumPy, a developer can perform the following operations −

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

NumPy – Environment

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

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(https://www.python-xy.github.io/)

Linux

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

For Ubuntu

sudo apt-get install python-numpy
python-scipy python-matplotlibipythonipythonnotebook python-pandas
python-sympy python-nose

For Fedora

sudo yum install numpyscipy python-matplotlibipython
python-pandas sympy python-nose atlas-devel

Building from Source

Core Python (2.6.x, 2.7.x and 3.2.x onwards) must be installed with distutils and zlib module should be enabled.

GNU gcc (4.2 and above) C compiler must be available.

To install NumPy, run the following command.

Python setup.py install

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

import numpy

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'

Alternatively, NumPy package is imported using the following syntax −

import numpy as np

NumPy - 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**).

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

Examples:

1.

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

The output is as follows −

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

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]]
```

3.

```
# 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]
```

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)

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
```

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]
```

NumPy - Array Attributes

Here, 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

```
# an array of evenly spaced numbers
import numpy as np
a = np.arange(24)
print a
```

The output is as follows −

```
[0 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 17 18 19 20 21 22 23]
```

Example 2

```
# this is one dimensional array
import numpy as np
a = np.arange(24)
a.ndim
# now reshape it
b = a.reshape(2,4,3)
print b
# b is having three dimensions
```

The output is as follows −

```
[[[ 0,  1,  2]
  [ 3,  4,  5]
  [ 6,  7,  8]
  [ 9, 10, 11]]
```

```
[[12, 13, 14]
 [15, 16, 17]
 [18, 19, 20]
 [21, 22, 23]]]
```

numpy.itemsize

This array attribute returns the length of each element of array in bytes.

Example 1

```
# dtype of array is int8 (1 byte)
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.int8)
print x.itemsize
```

The output is as follows −

```
1
```

Example 2

```
# dtype of array is now float32 (4 bytes)
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.float32)
print x.itemsize
```

The output is as follows −

```
4
```

numpy.flags

The ndarray object has the following attributes. Its current values are returned by this function.

Sr.No.	Attribute & Description
1	C_CONTIGUOUS (C) The data is in a single, C-style contiguous segment
2	F_CONTIGUOUS (F) The data is in a single, Fortran-style contiguous segment
3	OWNDATA (O) The array owns the memory it uses or borrows it from another object
4	WRITEABLE (W) The data area can be written to. Setting this to False locks the data, making it read-only
5	ALIGNED (A) The data and all elements are aligned appropriately for the hardware

6	UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array

Example

The following example shows the current values of flags.

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

The output is as follows −

```
C_CONTIGUOUS : True
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

NumPy - Array Creation Routines

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

numpy.empty

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

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

The constructor takes the following parameters.

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

Example

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

The output is as follows −

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

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

numpy.zeros

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

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

The constructor takes the following parameters.

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

Example 1

```
# 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 2

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

Now, the output would be as follows −

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

Example 3

```
# custom type
import numpy as np
x = np.zeros((2,2), dtype = [('x', 'i4'), ('y', 'i4')])
print x
```

It should produce the following output −

```
[[(0,0)(0,0)]
 [(0,0)(0,0)]]
```

numpy.ones

Returns a new array of specified size and type, filled with ones.
numpy.ones(shape, dtype = None, order = 'C')
The constructor takes the following parameters.

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

Example 1

```
# 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 2

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

Now, the output would be as follows −

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

NumPy - Array From Existing Data

Here, we will discuss how to create an 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

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.]
```

NumPy - Array From Numerical Ranges

Here, we will see how to create an 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.

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

Example 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 basestop
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

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 - Indexing & Slicing

- Contents of ndarray object can be accessed and modified by indexing or slicing
- As mentioned earlier, items in ndarray object follows zero-based index.
- Types of indexing methods are available − **basic slicing** and **advanced indexing**.

Basic slicing

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]
```

Its 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
```

Here, we will get the same output −

[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
```

Its output is as follows −

5

Example 4

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

Now, the output would be −

[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]
```

Here, the output would be −

[2 3 4]

The above description applies to multi-dimensional **ndarray** too.

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.

```
# 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 of this program 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]]
```

Advanced Slicing:

It is possible to make a selection from ndarray that is a non-tuple sequence, ndarray object of integer or Boolean data type, or a tuple with at least one item being a sequence object. Advanced indexing always returns a copy of the data.
There are two types of advanced indexing − **Integer** and **Boolean**.

Integer Indexing

This mechanism helps in selecting any arbitrary item in an array based on its Ndimensional index. Each integer array represents the number of indexes into that dimension. When the index consists of as many integer arrays as the dimensions of the target ndarray, it becomes straightforward.

In the following example, one element of specified column from each row of ndarray object is selected. Hence, the row index contains all row numbers, and the column index specifies the element to be selected.

Example 1

```
import numpy as np
x = np.array([[1, 2], [3, 4], [5, 6]])
y = x[[0,1,2], [0,1,0]]
print y
```

Its output would be as follows −

```
[1  4  5]
```

The selection includes elements at (0,0), (1,1) and (2,0) from the first array.

In the following example, elements placed at corners of a 4X3 array are selected. The row indices of selection are [0, 0] and [3,3] whereas the column indices are [0,2] and [0,2].
Example 2

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

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

rows = np.array([[0,0],[3,3]])
cols = np.array([[0,2],[0,2]])
y = x[rows,cols]

print 'The corner elements of this array are:'
print y
```

The output of this program is as follows −

```
Our array is:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

The corner elements of this array are:
[[ 0  2]
 [ 9 11]]
```

Boolean Array Indexing

This type of advanced indexing is used when the resultant object is meant to be the result of Boolean operations, such as comparison operators.

Example 1

In this example, items greater than 5 are returned as a result of Boolean indexing.

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

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

# Now we will print the items greater than 5
print 'The items greater than 5 are:'
```

```
print x[x > 5]
```

The output of this program would be −

```
Our array is:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

The items greater than 5 are:
[ 6  7  8  9 10 11]
```

Example 2

In this example, NaN (Not a Number) elements are omitted by using ~ (complement operator).

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

Its output would be −

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

Example 3

The following example shows how to filter out the non-complex elements from an array.

```
import numpy as np
a = np.array([1, 2+6j, 5, 3.5+5j])
print a[np.iscomplex(a)]
```

Here, the output is as follows −

```
[2.0+6.j  3.5+5.j]
```