UNIT – II

NumPy

NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

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.

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 - 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

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)

NumPy - Array Manipulation

Several routines are available in NumPy package for manipulation of elements in ndarray object. They can be classified into the following types −

Changing Shape

Sr.No.	Shape & Description
1	reshape Gives a new shape to an array without changing its data
2	flat A 1-D iterator over the array
3	flatten Returns a copy of the array collapsed into one dimension
4	ravel Returns a contiguous flattened array

Transpose Operations

Sr.No.	Operation & Description
1	transpose Permutes the dimensions of an array
2	ndarray.T Same as self.transpose()
3	rollaxis Rolls the specified axis backwards
4	swapaxes Interchanges the two axes of an array

Changing Dimensions

Sr.No.	Dimension & Description
1	broadcast Produces an object that mimics broadcasting
2	broadcast_to Broadcasts an array to a new shape
3	expand_dims Expands the shape of an array
4	squeeze Removes single-dimensional entries from the shape of an array

Joining Arrays

Sr.No.	Array & Description
1	concatenate Joins a sequence of arrays along an existing axis
2	stack Joins a sequence of arrays along a new axis
3	hstack Stacks arrays in sequence horizontally (column wise)
4	vstack Stacks arrays in sequence vertically (row wise)

Splitting Arrays

Sr.No.	Array & Description
1	split Splits an array into multiple sub-arrays
2	hsplit Splits an array into multiple sub-arrays horizontally (column-wise)
3	vsplit Splits an array into multiple sub-arrays vertically (row-wise)

Adding / Removing Elements

Sr.No.	Element & Description
1	resize Returns a new array with the specified shape
2	append Appends the values to the end of an array
3	insert Inserts the values along the given axis before the given indices
4	delete Returns a new array with sub-arrays along an axis deleted
5	unique Finds the unique elements of an array

NumPy - Binary 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

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
2	multiply()

	Returns the string with multiple concatenation, element-wise
3	center() Returns a copy of the given string with elements centered in a string of specified length
4	capitalize() Returns a copy of the string with only the first character capitalized
5	title() Returns the element-wise title cased version of the string or unicode
6	lower() Returns an array with the elements converted to lowercase
7	upper() Returns an array with the elements converted to uppercase
8	split() Returns a list of the words in the string, using separatordelimiter
9	splitlines() Returns a list of the lines in the element, breaking at the line boundaries
10	strip() Returns a copy with the leading and trailing characters removed
11	join() Returns a string which is the concatenation of the strings in the sequence

12	replace() Returns a copy of the string with all occurrences of substring replaced by the new string
13	decode() Calls str.decode element-wise
14	encode() Calls str.encode element-wise

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

Live Demo

```
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

Live Demo

```
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

Live Demo

```
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

Live Demo

```
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.]
```

NumPy - Arithmetic Operations

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

Example

Live Demo

```
import numpy as np
a = np.arange(9, dtype = np.float_).reshape(3,3)

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

print 'Second array:'
```

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

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

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

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

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

It will produce the following output −

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

Second array:
[10 10 10]

Add the two arrays:
[[ 10. 11. 12.]
 [ 13. 14. 15.]
 [ 16. 17. 18.]]

Subtract the two arrays:
```

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

Multiply the two arrays:
[[ 0. 10. 20.]
 [ 30. 40. 50.]
 [ 60. 70. 80.]]

Divide the two arrays:
[[ 0. 0.1 0.2]
 [ 0.3 0.4 0.5]
 [ 0.6 0.7 0.8]]
```

Let us now discuss some of the other important arithmetic functions available in NumPy.

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

Live Demo

```
import numpy as np
a = np.array([0.25, 1.33, 1, 0, 100])

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

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

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

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

It will produce the following output −

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

After applying reciprocal function:
main.py:9: RuntimeWarning: divide by zero encountered in reciprocal
  print np.reciprocal(a)
[ 4.         0.7518797  1.                inf  0.01     ]

The second array is:
[100]

After applying reciprocal function:
[0]
```

numpy.power()

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

Live Demo

```
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)
```

It will produce the following output −

```
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.

Live Demo

```
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)
```

It will produce the following output −

```
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.

Live Demo

```
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)
```

It will produce the following output −

```
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.]
```

NumPy - Statistical Functions

NumPy has quite a few useful statistical functions for finding minimum, maximum, percentile standard deviation and variance, etc. from the given elements in the array. The functions are explained as follows −

numpy.amin() and numpy.amax()

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

Example

Live Demo

```
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.

Live Demo

```
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