Day 11

Python for Data Science

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.
- It also discusses the various array functions, types of indexing, etc. An introduction to Matplotlib is also provided. All this is explained with the help of examples for better understanding.

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

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.

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

[1 2 3]

```
type(a)
```

numpy.ndarray

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

[[1 2]
 [3 4]]

- 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.

NumPy - Array Attributes

- ndarray.shape : This array attribute returns a tuple consisting of array dimensions. It can also be used to resize the array.

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

(2, 3)

- NumPy also provides a reshape function to resize an array.

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

(2, 3)
[[1 2 3]
 [4 5 6]]

(3, 2)
[[1 2]
 [3 4]
 [5 6]]

•ndarray.ndim : This array attribute returns the number of array dimensions.

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

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

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

```
1

[[[ 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 - 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

The following code shows an example of an empty array.

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

```
[[         0 1072693248]
 [         0 1073741824]
 [         0 1074266112]]
```

•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

```
# array of five zeros. Default dtype is float
import numpy as np
x = np.zeros(5)
print (x)
```

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

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

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

```
C:\Users\bytei\AppData\Local\Temp\ipykernel_13136\962979758.py:2: DeprecationWarning: `np.int` is a deprecated alias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x = np.zeros((5), dtype = np.int)
```

• 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

```
# array of five ones. Default dtype is float
import numpy as np
x = np.ones(5)
print (x)
```

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

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

[[1 1]
 [1 1]]

NumPy - Array From Existing Data

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

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

The constructor takes the following parameters.

Sr. No.	Parameter & Description
1	**a** Input data in any form 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

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

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

[1 2 3]

```
# dtype is set
import numpy as np

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

[1. 2. 3.]

```
# ndarray from tuple
import numpy as np

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

[1 2 3]

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

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

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

C:\Users\bytei\AppData\Local\Temp\ipykernel_13136\2789599881.py:5: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  a = np.asarray(x)

• 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)

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.

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

range(0, 5)

```
# obtain iterator object from list
import numpy as np
list1 = range(5)
it = iter(list1)

# use iterator to create ndarray
x = np.fromiter(it, dtype = float)
print (x)
```

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

NumPy - 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, default 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

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

[0 1 2 3 4]

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

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

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

[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 start 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

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

[10. 12.5 15. 17.5 20.]

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

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

```
# find retstep value
import numpy as np

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

(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 start point of the sequence is $base^{start}$
2	**stop** The final value of sequence is $base^{stop}$
3	**num** The number of value 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

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

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

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

[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, 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.

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

```
[0 1 2 3 4 5 6 7 8 9]
[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.

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

```
[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.

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

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

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

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

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

```
[2 3 4]
```

```
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:])
```

```
[[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:])
```

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]]

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.

```
import numpy as np

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

[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.

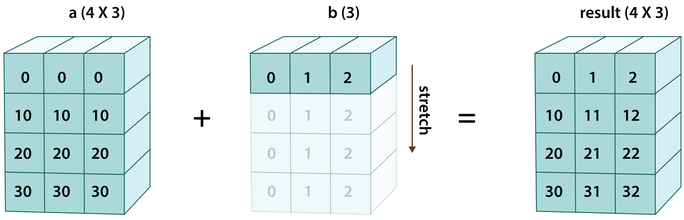
```
import numpy as np
a = np.ar-
ray([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],[30.0,30.0,30.
0]])
b = np.array([1.0,2.0,3.0])

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

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

print ('First Array + Second Array')
print (a + b)
```

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 - Iterating Over Array

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.

```
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),
```

```
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
```

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),
```

```
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.

```
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),
```

Output exceeds the size limit. Open the full output data in a text editor
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
...
40
45
50
55

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),
```

Output exceeds the size limit. Open the full output data in a text editor
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
...
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.

```
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)
```

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]]

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).

```
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)),
```

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 - 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	**roliaxis** 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 shap
3	**expand_dims** Expands the shape of an array
4	**squeeze** Removes single-dimensional entires 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.	Elements & 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 a binary representation to the right

NumPy - String Functions

Sr. No.	Operation & 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()** Cals str.decode element wise
14	**encode()** Cals 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.

```
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))
```

```
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]
```

Functions for Rounding

• numpy.around() :This is a function that returns the value rounded to the desired precision. The function takes the following parameters.

numpy.around(a,decimals)

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

```
import numpy as np
a = np.array([1.0,5.55, 123, 0.567, 25.532])

print ('Original array:')
print (a)
print ('\n')

print ('After rounding:')
print (np.around(a))
print (np.around(a, decimals = 1))
print (np.around(a, decimals = -1))
```

Original array:
[1. 5.55 123. 0.567 25.532]

After rounding:
[1. 6. 123. 1. 26.]
[1. 5.6 123. 0.6 25.5]
[0. 10. 120. 0. 30.]

• numpy.floor() :This function returns the largest integer not greater than the input parameter. The floor of the scalar x is the largest integer i, such that i <= x. Note that in Python, flooring always is rounded away from 0.

```
import numpy as np
a = np.array([-1.7, 1.5, -0.2, 0.6, 10])

print ('The given array:')
print (a)
print ('\n')

print ('The modified array:')
print (np.floor(a))
```

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

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

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

```
import numpy as np
a = np.array([-1.7, 1.5, -0.2, 0.6, 10])

print ('The given array:')
print (a)
print ('\n')

print ('The modified array:')
print (np.ceil(a))
```

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

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

NumPy - Arithmetic Operations

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

```
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))
```

Output exceeds the size limit. Open the full output data in a text editor
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.]
...
Divide the two arrays:
[[0. 0.1 0.2]
 [0.3 0.4 0.5]
 [0.6 0.7 0.8]]

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

```
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))
```

- 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.

```
import numpy as np
a = np.array([[30,65,70],[80,95,10],[50,90,60]])

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

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

print ('Applying median() function along axis 0:')
print (np.median(a, axis = 0))
print ('\n')

print ('Applying median() function along axis 1:')
print (np.median(a, axis = 1))
```

```
Our array is:
[[30 65 70]
 [80 95 10]
 [50 90 60]]


Applying median() function:
65.0


Applying median() function along axis 0:
[50. 90. 60.]


Applying median() function along axis 1:
[65. 80. 60.]
```

- numpy.mean()

Arithmetic mean is the sum of elements along an axis divided by the number of elements. The numpy.mean() function returns the arithmetic mean of elements in the array. If the axis is mentioned, it is calculated along it.

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

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

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

print ('Applying mean() function along axis 0:')
print (np.mean(a, axis = 0))
print ('\n')

print ('Applying mean() function along axis 1:')
print (np.mean(a, axis = 1))
```

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


Applying mean() function:
3.6666666666666665


Applying mean() function along axis 0:
[2.66666667 3.66666667 4.66666667]


Applying mean() function along axis 1:
[2. 4. 5.]
```

numpy.average()

Weighted average is an average resulting from the multiplication of each component by a factor reflecting its importance. The numpy.average() function computes the weighted average of elements in an array according to their respective weight given in another array. The function can have an axis parameter. If the axis is not specified, the array is flattened.

Considering an array [1,2,3,4] and corresponding weights [4,3,2,1], the weighted average is calculated by adding the product of the corresponding elements and dividing the sum by the sum of weights.

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

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

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

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

# this is same as mean when weight is not specified
wts = np.array([4,3,2,1])

print ('Applying average() function again:')
print (np.average(a,weights = wts))
print ('\n')

# Returns the sum of weights, if the returned parameter is set to True.
print ('Sum of weights')
print (np.average([1,2,3, 4],weights = [4,3,2,1], returned = True))
```

In a multi-dimensional array, the axis for computation can be specified.

```
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))
```

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.

```
import numpy as np
print (np.std([1,2,3,4]))
```

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.

```
import numpy as np
print (np.var([1,2,3,4]))
```

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() & numpy.argmin()

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

```
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)
```

Output exceeds the size limit. Open the full output data in a text editor
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 along axis 1:
[0 2 0]