TwoWaits Internship - Tathasthu Data Science & Machine Learning Scholar Intern

TASK 1 - BASICS OF NUMPY AND PANDAS

Name: Tanushree B S

ID : TSM_0708418

NUMPY - BASICS

- Python package
- Stands for 'Numerical Python'
- Open source platform
- Using NumPy, a developer can perform the following operations:

  (a) Mathematical and logical operations on arrays.

  (b) Fourier transforms and routines for shape manipulation.

  (c) Operations related to linear algebra and random number generation.

1. NumPy - ndarray object

- NumPy package is imported using the following syntax:

  import numpy as np
- Most important object defined in NumPy is an N-dimensional array type called ndarray. It describes the collection of items of the same type.
- 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)

- Syntax of creating an array

```
numpy.array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)
```

In [1]:

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

```
[1 2 3]
```

In [6]:

```
# more than one dimensional array
a=np.array([[1,2,3],[4,5,6]])
print(a)
```

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

In [10]:

```
# minimum dimensions
a=np.array([1,2,3,4,5],ndmin=2) #ndmin- specifies the dimension of resultant array
print(a)
```

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

In [12]:

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

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

2. NumPy- Array Attributes

ndarray.shape

- returns a tuple consisting of array dimensions.
- can also be used to resize the array.

In [18]:

```
#gives the dimensions of array
a=np.array([[1,2,3],[4,5,6]])
print(a.shape)
print(a)
```

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

In [19]:

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

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

In [23]:

```
#reshapes function - to resize an array
a=np.array([[1,2,3],[4,5,6]])
b=a.reshape(3,2)
print(b)
```

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

ndarray.ndim

- returns the number of array dimensions.

In [25]:

```
a=np.array([[1,2,3],[4,5,6]])
print(a.ndim)
```

```
2
```

In [27]:

```
a=np.array([1,2,3])
print(a.ndim)
```

```
1
```

In [32]:

```
a=np.arange(24)
b=a.reshape(2,4,3)
print(b) #b is having three dimensions
```

```
[[[ 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

- returns the length of each element of array in bytes.

In [35]:

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

```
1
```

In [37]:

```
# 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)
```

```
4
```

3.NumPy-Array Creation Routines

- new ndarray object can be constructed by any of the following array creation routines

numpy.empty

- creates an uninitialized array of specified shape and dtype.
- Syntax:

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

In [47]:

```
x = np.empty((3,2), dtype=int)
print(x) #the values which are obtained in the output is random values
```

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

numpy.zeroes

- returns a new array of specified size, filled with zeros.
- Syntax:

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

In [43]:

```
x = np.zeros(5)
print (x)
```

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

In [49]:

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

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

In [52]:

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

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

numpy.ones

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

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

In [54]:

```
x = np.ones(5)
print(x)
```

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

In [58]:

```
x = np.ones((2,2), dtype=int)
print (x)
```

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

In [59]:

```
#custom type
x = np.ones((2,2), dtype=[('x', 'i4'), ('y', 'i4')])
print(x)
```

```
[[(1, 1) (1, 1)]
 [(1, 1) (1, 1)]]
```

4.NumPy- Array from Existing data

numpy.asarray

- useful for converting Python sequence into ndarray.
- Syntax:

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

In [4]:

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

```
[1 2 3]
```

In [6]:

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

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

In [8]:

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

```
[1 2 3]
```

In [10]:

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

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

numpy.frombuffer

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

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

In [13]:

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

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

numpy.fromiter

- builds an ndarray object from any iterable object.
- A new one-dimensional array is returned.
- Syntax:

```
numpy.fromiter(iterable, dtype, count=-1)
```

In [19]:

```
list=range(5)
it=iter(list)
x=np.fromiter(it,dtype=float)
print(x)
```

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

5.NumPy- Array from Numerical Ranges

numpy.arange

- returns an ndarray object containing evenly spaced values within a given range.
- Syntax:

```
numpy.arange(start, stop, step, dtype)
```

In [21]:

```
x = np.arange(5)
print (x)
```

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

In [23]:

```
x = np.arange(5, dtype=float)
print(x)
```

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

In [25]:

```
x = np.arange(10,20,2)
print (x)
```

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

numpy.linspace

- instead of step size, the number of evenly spaced values between the interval is specified.
- Syntax:

```
numpy.linspace(start, stop, num, endpoint, retstep, dtype)
```

In [27]:

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

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

In [29]:

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

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

In [32]:

```
# 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

- 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.
- Syntax:

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

In [34]:

```
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.        ]
```

In [36]:

```
a = np.logspace(1,10,num=10, base=2)
print (a)
```

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

6.NumPy- Indexing and Slicing

- Three types of indexing methods are available: field access, basic slicing and advanced indexing.

In [3]:

```
import numpy as np
a = np.arange(10)
# a= 1 2 3 4 5 6 7 8 9 10
s = slice(2,7,2) #slice object is defined with start, stop, and step values 2, 7, and 2 res
# s= 2 4 6
print(s)
print (a[s])
```

```
slice(2, 7, 2)
[2 4 6]
```

In [5]:

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

```
[2 4 6]
```

In [10]:

```
#slicing single item
a = np.arange(10)
print(a)
b = a[5]
print (b)
```

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

In [13]:

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

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

In [15]:

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

```
[2 3 4]
```

In [22]:

```
#slicing in multi-dimensional array
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print (a)
# slice items starting from indexNumPy
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.

In [19]:

```
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print(a)
# this returns array of items in the second column
print("\n",a[...,1])
# Now we will slice all items from the second row
print("\n",a[1,...])
# Now we will slice all items from column 1 onwards
print("\n",a[...,1:])
#will print the whole contents of the array
print("\n",a[...])
```

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

 [2 4 5]

 [3 4 5]

 [[2 3]
 [4 5]
 [5 6]]

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

7.NumPy- Advanced Indexing

- There are two types of advanced indexing: Integer and Boolean.
- **INTERGER INDEXING**

In [34]:

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

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

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

In [40]:

```
x = np.array([[ 0, 1, 2],[ 3, 4, 5],[ 6, 7, 8],[ 9, 10, 11]])
print(x,"\n")
rows = np.array([[0,0],[3,3]])
print(rows,"\n")
cols = np.array([[0,2],[0,2]])
print(cols,"\n")
y=x[rows,cols]
print(y)
```

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

[[0 0]
 [3 3]]

[[0 2]
 [0 2]]

[[ 0  2]
 [ 9 11]]
```

In [47]:

```
x = np.array([[0, 1, 2],[3, 4, 5],[6, 7, 8],[9, 10, 11]])
print(x)
# slicing
z = x[1:4,1:3]
print ('After slicing, our array becomes:')
print(z)

# using advanced index for column
y = x[1:4,[1,2]]
print('Slicing using advanced index for column:')
print(y)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
After slicing, our array becomes:
[[ 4  5]
 [ 7  8]
 [10 11]]
Slicing using advanced index for column:
[[ 4  5]
 [ 7  8]
 [10 11]]
```

- **BOOLEAN ARRAY INDEXING**

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

Ex 1:Items greater than 5 are returned as a result of Boolean indexing.

In [50]:

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

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

Ex 2:NaN (Not a Number) elements are omitted by using ~ (complement operator).

In [56]:

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

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

Ex 3:filter out the non-complex elements from an array.

In [54]:

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

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

8.NumPy- Broadcasting

- The ability of NumPy to treat arrays of different shapes during arithmetic operations.
- If two arrays are of exactly the same shape, then these operations are smoothly performed.

In [2]:

```
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.

Following should be true for a set of arrays to be broadcastable:

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

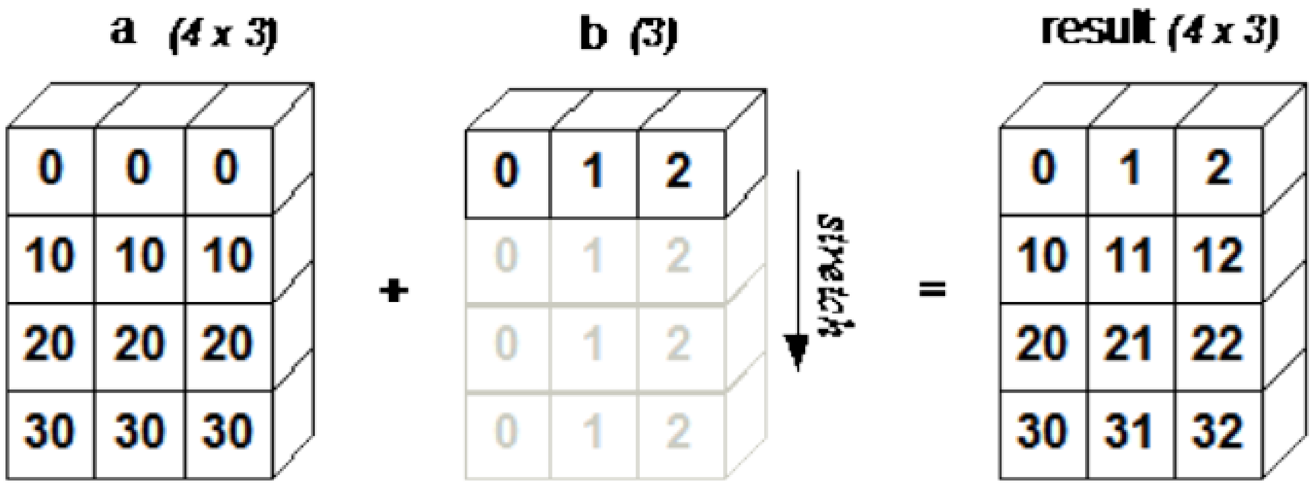
In [13]:

```
a = np.array([[ 0.0, 0.0, 0.0],[10.0,10.0,10.0],
 [20.0,20.0,20.0],[30.0,30.0,30.0]])
b = np.array([1.0,2.0,3.0])
print(a)
print(a.shape,"\n")
print(b)
print(b.shape,"\n")
print(a+b)
```

```
[[ 0.  0.  0.]
 [10. 10. 10.]
 [20. 20. 20.]
 [30. 30. 30.]]
(4, 3)

[1. 2. 3.]
(3,)

[[ 1.  2.  3.]
 [11. 12. 13.]
 [21. 22. 23.]
 [31. 32. 33.]]
```



9.NumPy-Iterating over array

- numpy.nditer:

efficient multi-dimensional iterator object using which it is possible to iterate over an array.

In [2]:

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

In [7]:

```
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

In [9]:

```
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:') #would be operating row wise
c = b.copy(order='C')
print (c)
for x in np.nditer(c):
    print (x)
print ('\n')
print ('Sorted in F-style order:') #would be operating column wise
c = b.copy(order='F')
print (c)
for x in np.nditer(c):
    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]]


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


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

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

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

In [11]:

```
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)
```

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


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


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

MODIFIED 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.

In [22]:

```
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]]
```

EXTERNAL FLAGS

The nditer class constructor has a ‘flags’ parameter, which can take the following values:

c_index	C_order index can be racked
f_index	Fortran_order index is tracked
multi-index	Type of indexes with one per iteration can be tracked
external_loop	Causes values given to be one-dimensional arrays with multiple values instead of zero-dimensional array

In [24]:

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print ('Original array is:')
print (a)
print ('\n')
print ('Modified array is:')
for x in np.nditer(a, flags=['external_loop'], order='F'):
    print (x)
```

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


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

BROADCASTING ITERATION

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

In [28]:

```
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
```

10.NumPy-Array Manipulation

Changing Shape

reshape	Gives a new shape to an array without changing its data
flat	A 1-D iterator over the array
flatten	Returns a copy of the array collapsed into one dimension
ravel	Returns a contiguous flattened array

Transpose Operations

transpose	Permutes the dimensions of an array
ndarray.T	Same as self.transpose()
rollaxis	Rolls the specified axis backwards
swapaxes	Interchanges the two axes of an array

Changing Dimensions

broadcast	Produces an object that mimics broadcasting
broadcast_to	Broadcasts an array to a new shape
expand_dims	Expands the shape of an array
squeeze	Removes single-dimensional entries from the shape of an array

Joining Arrays

concatenate	Joins a sequence of arrays along an existing axis
stack	Joins a sequence of arrays along a new axis
hstack	Stacks arrays in sequence horizontally (column wise)
vstack	Stacks arrays in sequence vertically (row wise)

Splitting Arrays

split	Splits an array into multiple sub-arrays
hsplit	Splits an array into multiple sub-arrays horizontally (column-wise)
vsplit	Splits an array into multiple sub-arrays vertically (row-wise)

Adding / Removing Elements

resize	Returns a new array with the specified shape
append	Appends the values to the end of an array
insert	Inserts the values along the given axis before the given indices
delete	Returns a new array with sub-arrays along an axis deleted
unique	Finds the unique elements of an array

- **numpy.reshape**

Gives a new shape to an array without changing the data. It accepts the following parameters:

numpy.reshape(arr, newshape, order')

In [30]:

```
import numpy as np
a = np.arange(8)
print ('The original array:')
print (a)
print ('\n')
b = a.reshape(4,2)
print ('The modified array:')
print (b)
```

```
The original array:
[0 1 2 3 4 5 6 7]


The modified array:
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
```

- **numpy.ndarray.flat**

returns a 1-D iterator over the array

In [32]:

```
a = np.arange(8).reshape(2,4)
print ('The original array:')
print (a)
print ('\n')
print ('After applying the flat function:')
# returns element corresponding to index in flattened array
print (a.flat[5])
```

```
The original array:
[[0 1 2 3]
 [4 5 6 7]]


After applying the flat function:
5
```

- **numpy.ndarray.flatten**

  returns a copy of an array collapsed into one dimension. The function takes the following parameters:

  ndarray.flatten(order)

In [35]:

```
import numpy as np
a = np.arange(8).reshape(2,4)
print ('The original array is:')
print (a)
print ('\n')
# default is column-major
print ('The flattened array is:')
print (a.flatten())
print ('\n')
print ('The flattened array in F-style ordering:')
print (a.flatten(order='F'))
"""'C': row major (default. 'F': column major 'A': flatten in column-major
order, if a is Fortran contiguous in memory, row-major
order otherwise 'K': flatten a in the order the elements
occur in the memory"""
```

```
The original array is:
[[0 1 2 3]
 [4 5 6 7]]


The flattened array is:
[0 1 2 3 4 5 6 7]


The flattened array in F-style ordering:
[0 4 1 5 2 6 3 7]
```

Out[35]:

```
"'C': row major (default. 'F': column major 'A': flatten in column-major \no
rder, if a is Fortran contiguous in memory, row-major \norder otherwise 'K':
flatten a in the order the elements \noccur in the memory"
```

- **numpy.ravel**

  returns a flattened one-dimensional array. A copy is made only if needed. The returned array will have the same type as that of the input array.

  Syntax:

  numpy.ravel(a, order)

In [38]:

```
a = np.arange(8).reshape(2,4)
print ('The original array is:')
print (a)
print ('\n')
print ('After applying ravel function:')
print (a.ravel())
print ('\n')
print ('Applying ravel function in F-style ordering:')
print (a.ravel(order='F'))
```

```
The original array is:
[[0 1 2 3]
 [4 5 6 7]]


After applying ravel function:
[0 1 2 3 4 5 6 7]


Applying ravel function in F-style ordering:
[0 4 1 5 2 6 3 7]
```

- **numpy.transpose**

This function permutes the dimension of the given array. It returns a view wherever possible. The function takes the following parameters.

numpy.transpose(arr, axes)

In [40]:

```
a = np.arange(12).reshape(3,4)
print ('The original array is:')
print (a)
print ('\n')
print ('The transposed array is:')
print (np.transpose(a))
```

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


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

- **numpy.ndarray.T**

This function belongs to ndarray class. It behaves similar to numpy.transpose.

In [42]:

```
a = np.arange(12).reshape(3,4)
print ('The original array is:')
print (a)
print ('\n')
print ('Array after applying the function:')
print (a.T)
```

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


Array after applying the function:
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

- **numpy.swapaxes**

This function interchanges the two axes of an array. The function takes the following parameters.

numpy.swapaxes(arr, axis1, axis2)

In [45]:

```
a = np.arange(8).reshape(2,2,2)
print ('The original array:')
print (a)
print ('\n')
# now swap numbers between axis 0 (along depth) and axis 2 (along width)
print ('The array after applying the swapaxes function:')
print (np.swapaxes(a, 2, 0))
```

```
The original array:
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]


The array after applying the swapaxes function:
[[[0 4]
  [2 6]]

 [[1 5]
  [3 7]]]
```

- **numpy.rollaxis**

This function rolls the specified axis backwards, until it lies in a specified position. The function takes three parameters.

numpy.rollaxis(arr, axis, start)

In [48]:

```
a = np.arange(8).reshape(2,2,2)
print ('The original array:')
print (a)
print ('\n')
# to roll axis-2 to axis-0 (along width to along depth)
print ('After applying rollaxis function:')
print (np.rollaxis(a,2))
# to roll axis 0 to 1 (along width to height)
print ('\n')
print ('After applying rollaxis function:')
print (np.rollaxis(a,2,1))
```

```
The original array:
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]


After applying rollaxis function:
[[[0 2]
  [4 6]]

 [[1 3]
  [5 7]]]


After applying rollaxis function:
[[[0 2]
  [1 3]]

 [[4 6]
  [5 7]]]
```

- **numpy.broadcast_to**

It returns a read-only view on the original array. It is typically not contiguous. The function may throw ValueError if the new shape does not comply with NumPy's broadcasting rules. The function takes the following parameters.

numpy.broadcast_to(array, shape, subok)

In [53]:

```
a = np.arange(4).reshape(1,4)
print ('The original array:')
print (a)
print ('\n')
print ('After applying the broadcast_to function:')
print (np.broadcast_to(a,(4,4)))
```

```
The original array:
[[0 1 2 3]]


After applying the broadcast_to function:
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
```

- **numpy.expand_dims**

This function expands the array by inserting a new axis at the specified position. Two parameters are required by this function.

numpy.expand_dims(arr, axis)

In [56]:

```
x = np.array(([1,2],[3,4]))
print ('Array x:')
print (x)
print ('\n')
y = np.expand_dims(x, axis=0)
print ('Array y:')
print (y)
print ('\n')
print ('The shape of X and Y array:')
print (x.shape, y.shape)
print ('\n')
# insert axis at position 1
y = np.expand_dims(x, axis=1)
print ('Array Y after inserting axis at position 1:')
print (y)
print ('\n')
print ('x.ndim and y.ndim:')
print (x.ndim,y.ndim)
print ('\n')
print ('x.shape and y.shape:')
print (x.shape, y.shape)
```

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


Array y:
[[[1 2]
  [3 4]]]


The shape of X and Y array:
(2, 2) (1, 2, 2)


Array Y after inserting axis at position 1:
[[[1 2]]

 [[3 4]]]


x.ndim and y.ndim:
2 3


x.shape and y.shape:
(2, 2) (2, 1, 2)
```

- **numpy.squeeze**

This function removes one-dimensional entry from the shape of the given array. Two parameters are required for this function.

numpy.squeeze(arr, axis)

In [58]:

```
x = np.arange(9).reshape(1,3,3)
print ('Array X:')
print (x)
print ('\n')
y = np.squeeze(x)
print ('Array Y:')
print (y)
print ('\n')
print ('The shapes of X and Y array:')
print (x.shape, y.shape)
```

```
Array X:
[[[0 1 2]
  [3 4 5]
  [6 7 8]]]


Array Y:
[[0 1 2]
 [3 4 5]
 [6 7 8]]


The shapes of X and Y array:
(1, 3, 3) (3, 3)
```

- numpy.concatenate

used to join two or more arrays of the same shape along a specified axis. The function takes the following parameters.

numpy.concatenate((a1, a2, ...), axis)

In [60]:

```
a=np.array([[1,2],[3,4]])
print ('First array:')
print (a)
print ('\n')
b = np.array([[5,6],[7,8]])
print ('Second array:')
print (b)
print ('\n')
# both the arrays are of same dimensions
print ('Joining the two arrays along axis 0:')
print (np.concatenate((a,b)))
print ('\n')
print ('Joining the two arrays along axis 1:')
print (np.concatenate((a,b),axis=1))
```

```
First array:
[[1 2]
 [3 4]]


Second array:
[[5 6]
 [7 8]]


Joining the two arrays along axis 0:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]


Joining the two arrays along axis 1:
[[1 2 5 6]
 [3 4 7 8]]
```

- **numpy.stack**

joins the sequence of arrays along a new axis.Following parameters need to be provided.

numpy.stack(arrays, axis)

In [63]:

```
a = np.array([[1,2],[3,4]])
print ('First Array:')
print (a)
print ('\n')
b = np.array([[5,6],[7,8]])
print ('Second Array:')
print (b)
print ('\n')
print ('Stack the two arrays along axis 0:') #axis 0- row wise
print (np.stack((a,b),0))
print ('\n')
print ('Stack the two arrays along axis 1:') #axis 1 - column wise
print (np.stack((a,b),1))
```

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


Second Array:
[[5 6]
 [7 8]]


Stack the two arrays along axis 0:
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]


Stack the two arrays along axis 1:
[[[1 2]
  [5 6]]

 [[3 4]
  [7 8]]]
```

- **numpy.hstack and numpy.vstack**

Variants of numpy.stack function to stack so as to make a single array horizontally or vertically.

In [66]:

```
a = np.array([[1,2],[3,4]])
print ('First array:')
print (a)
print ('\n')
b = np.array([[5,6],[7,8]])
print ('Second array:')
print (b)
print ('\n')
print ('Horizontal stacking:')
c = np.hstack((a,b))
print (c)
print ('\n')
print ('Vertical stacking:')
c = np.vstack((a,b))
print (c)
```

```
First array:
[[1 2]
 [3 4]]


Second array:
[[5 6]
 [7 8]]


Horizontal stacking:
[[1 2 5 6]
 [3 4 7 8]]


Vertical stacking:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

- **numpy.split()**

This function divides the array into subarrays along a specified axis. The function takes three parameters.

numpy.split(ary, indices_or_sections, axis)

In [68]:

```
a = np.arange(9)
print ('First array:')
print (a)
print ('\n')
print ('Split the array in 3 equal-sized subarrays:')
b = np.split(a,3)
print (b)
print ('\n')
print ('Split the array at positions indicated in 1-D array:')
b = np.split(a,[4,7])
print (b)
```

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


Split the array in 3 equal-sized subarrays:
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]


Split the array at positions indicated in 1-D array:
[array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8])]
```

- **numpy.hsplit and numpy.vsplit**

The numpy.hsplit is a special case of split() function where axis is 1 indicating a horizontal split regardless of the dimension of the input array. Similarly, numpy.vsplit is a special case of split() function where axis is 1 indicating a vertical split regardless of the dimension of the input array.

In [70]:

```
a = np.arange(16).reshape(4,4)
print ('First array:')
print (a)
print ('\n')
print ('Horizontal splitting:')
b = np.hsplit(a,2)
print (b)
print ('\n')
print ('Vertical splitting:')
b = np.vsplit(a,2)
print (b)
```

```
First array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]


Horizontal splitting:
[array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]]), array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])]


Vertical splitting:
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])]
```

- **numpy.resize**

  returns a new array with the specified size. If the new size is greater than the original, the repeated copies of entries in the original are contained. The function takes the following parameters.

  ```
  numpy.resize(arr, shape)
  ```

In [76]:

```
a = np.array([[1,2,3],[4,5,6]])
print ('First array:')
print (a)
print ('\n')
print ('The shape of first array:')
print (a.shape)
print ('\n')
b = np.resize(a, (3,2))
print ('Second array:')
print (b)
print ('\n')
print ('The shape of second array:')
print (b.shape)
print ('\n')
# Observe that first row of a is repeated in b since size is bigger
print ('Resize the second array:')
b = np.resize(a,(3,3))
print (b)
```

```
First array:
[[1 2 3]
 [4 5 6]]


The shape of first array:
(2, 3)


Second array:
[[1 2]
 [3 4]
 [5 6]]


The shape of second array:
(3, 2)


Resize the second array:
[[1 2 3]
 [4 5 6]
 [1 2 3]]
```

- **numpy.append**

adds values at the end of an input array. Also the dimensions of the input arrays must match otherwise ValueError will be generated.

In [2]:

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print ('First array:')
print (a)
print ('\n')
print ('Append elements to array:')
print (np.append(a, [7,8,9]))
print ('\n')
print ('Append elements along axis 0:')
print (np.append(a, [[7,8,9]],axis=0))
print ('\n')
print ('Append elements along axis 1:')
print (np.append(a, [[5,5,5],[7,8,9]],axis=1))
```

```
First array:
[[1 2 3]
 [4 5 6]]


Append elements to array:
[1 2 3 4 5 6 7 8 9]


Append elements along axis 0:
[[1 2 3]
 [4 5 6]
 [7 8 9]]


Append elements along axis 1:
[[1 2 3 5 5 5]
 [4 5 6 7 8 9]]
```

- **numpy.insert**

  inserts values in the input array along the given axis and before the given index. Also, if the axis is not mentioned, the input array is flattened. The insert() function takes the following parameters:

  numpy.insert(arr, obj, values, axis)

In [3]:

```
a = np.array([[1,2],[3,4],[5,6]])
print ('First array:')
print (a)
print ('\n')
print ('Axis parameter not passed. The input array is flattened before insertion.')
print (np.insert(a,3,[11,12]))
print ('\n')
print ('Axis parameter passed. The values array is broadcast to match input array.')
print ('Broadcast along axis 0:')
print (np.insert(a,1,[11],axis=0))
print ('\n')
print ('Broadcast along axis 1:')
print (np.insert(a,1,11,axis=1))
```

```
First array:
[[1 2]
 [3 4]
 [5 6]]


Axis parameter not passed. The input array is flattened before insertion.
[ 1  2  3 11 12  4  5  6]


Axis parameter passed. The values array is broadcast to match input array.
Broadcast along axis 0:
[[ 1  2]
 [11 11]
 [ 3  4]
 [ 5  6]]


Broadcast along axis 1:
[[ 1 11  2]
 [ 3 11  4]
 [ 5 11  6]]
```

- **numpy.delete**

returns a new array with the specified subarray deleted from the input array. Syntax:

numpy.delete(arr, obj, axis)

In [8]:

```
a = np.arange(12).reshape(3,4)
print ('First array:')
print (a)
print ('\n')
print ('Array flattened before delete operation as axis not used:')
print (np.delete(a,5))
print ('\n')
print ('Column 2 deleted:')
print (np.delete(a,1,axis=1))
print ('\n')
print ('A slice containing alternate values from array deleted:')
a = np.array([1,2,3,4,5,6,7,8,9,10])
print (np.delete(a, np.s_[::2]))
```

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


Array flattened before delete operation as axis not used:
[ 0  1  2  3  4  6  7  8  9 10 11]


Column 2 deleted:
[[ 0  2  3]
 [ 4  6  7]
 [ 8 10 11]]


A slice containing alternate values from array deleted:
[ 2  4  6  8 10]
```

- **numpy.unique**

returns an array of unique elements in the input array.

Syntax:

numpy.unique(arr, return_index, return_inverse, return_counts)

In [11]:

```
a = np.array([5,2,6,2,7,5,6,8,2,9])
print ('First array:')
print (a)
print ('\n')
print ('Unique values of first array:')
u = np.unique(a)
print (u)
print ('\n')
print ('Unique array and Indices array:')
u,indices=np.unique(a, return_index=True)
print (indices)
print ('\n')
print ('We can see each number corresponds to index in original array:')
print (a)
print ('\n')
print ('Indices of unique array:')
u,indices=np.unique(a,return_inverse=True)
print (u)
print ('\n')
print ('Indices are:')
print (indices)
print ('\n')
print ('Reconstruct the original array using indices:')
print (u[indices])
print ('\n')
print ('Return the count of repetitions of unique elements:')
u,indices=np.unique(a,return_counts=True)
print (u)
```

```
First array:
[5 2 6 2 7 5 6 8 2 9]


Unique values of first array:
[2 5 6 7 8 9]


Unique array and Indices array:
[1 0 2 4 7 9]


We can see each number corresponds to index in original array:
[5 2 6 2 7 5 6 8 2 9]


Indices of unique array:
[2 5 6 7 8 9]


Indices are:
[1 0 2 0 3 1 2 4 0 5]


Reconstruct the original array using indices:
[5 2 6 2 7 5 6 8 2 9]
```

```
Return the count of repetitions of unique elements:
[2 5 6 7 8 9]
```

11.NumPy- String Functions

add()	nent-wise string concatenation for two arrays of str or Unicode
multiply()	string with multiple concatenation, element-wise
center()	opy of the given string with elements centered in a string of specified length
capitalize()	py of the string with only the first character capitalized
title()	element-wise title cased version of the string or unicode
lower()	rray with the elements converted to lowercase
upper()	rray with the elements converted to uppercase
split()	t of the words in the string, using separatordelimiter
splitlines()	t of the lines in the element, breaking at the line boundaries
strip()	py with the leading and trailing characters removed
join()	ring which is the concatenation of the strings in the sequence
replace()	opy of the string with all occurrences of substring replaced by the new string
decode()	*ode* element-wise
encode()	*ode* element-wise

- **numpy.char.add()**

string concatenation

In [13]:

```
print ('Concatenate two strings:')
print (np.char.add(['hello'],[' xyz']))
print ('\n')
print ('Concatenation example:')
print (np.char.add(['hello', 'hi'],[' abc', ' xyz']))
```

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


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

- **numpy.char.multiply**

performs multiple concatenation

In [15]:

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

```
Hello Hello Hello
```

- **numpy.char.center()**

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

In [18]:

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

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

- **numpy.char.capitalize()**

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

In [20]:

```
print (np.char.capitalize('hello world'))
```

```
Hello world
```

- **numpy.char.title()**

returns a title cased version of the input string with the first letter of each word capitalized.

In [23]:

```
print (np.char.title('hello how are you?'))
```

```
Hello How Are You?
```

- **numpy.char.lower()**

returns an array with elements converted to lowercase. It calls str.lower for each element.

In [25]:

```
print (np.char.lower(['HELLO','WORLD']))
print (np.char.lower('HELLO'))
```

```
['hello' 'world']
hello
```

- **numpy.char.upper()**

calls str.upper function on each element in an array to return the uppercase array elements.

In [27]:

```
print (np.char.upper('hello'))
print (np.char.upper(['hello','world']))
```

```
HELLO
['HELLO' 'WORLD']
```

- **numpy.char.split()**

returns a list of words in the input string. By default, a whitespace is used as a separator. Otherwise the specified separator character is used to spilt the string.

In [29]:

```
print (np.char.split ('hello how are you?'))
print (np.char.split ('TutorialsPoint,Hyderabad,Telangana', sep=','))
```

```
['hello', 'how', 'are', 'you?']
['TutorialsPoint', 'Hyderabad', 'Telangana']
```

- **numpy.char.splitlines()**

returns a list of elements in the array, breaking at line boundaries.

In [31]:

```
print (np.char.splitlines('hello\nhow are you?'))
print (np.char.splitlines('hello\rhow are you?'))
```

```
['hello', 'how are you?']
['hello', 'how are you?']
```

- **numpy.char.strip()**

returns a copy of array with elements stripped of the specified characters leading and/or trailing in it.

In [33]:

```
print (np.char.strip('ashok arora','a'))
print (np.char.strip(['arora','admin','java'],'a'))
```

```
shok aror
['ror' 'dmin' 'jav']
```

- **numpy.char.join()**

returns a string in which the individual characters are joined by separator character specified.

In [36]:

```
print (np.char.join(':','dmy'))
print (np.char.join([':','-'],['dmy','ymd']))
```

```
d:m:y
['d:m:y' 'y-m-d']
```

- **numpy.char.replace()**

  returns a new copy of the input string in which all occurrences of the sequence of characters is replaced by another given sequence.

In [39]:

```
print (np.char.replace('He is a good boy', 'is', 'was'))
```

```
He was a good boy
```

- **numpy.char.encode()**

function calls str.encode function for each element in the array. Default encoding is utf_8, codecs available in standard Python library may be used. On the other hand, numpy.char.decode() decodes the given string using the specified codec.

In [43]:

```
a = np.char.encode('hello', 'cp500')
print (a)
print (np.char.decode(a,'cp500'))
```

```
b'\x88\x85\x93\x93\x96'
hello
```

12.NumPy-Arithmetic Operations

In [46]:

```
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))
```

```
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()**

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.

In [48]:

```
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))
```

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


After applying reciprocal function:
[4.        0.7518797 1.              inf 0.01     ]


The second array is:
[100]


After applying reciprocal function:
[0]

<ipython-input-48-ec6fa9109355>:6: RuntimeWarning: divide by zero encountere
d in reciprocal
  print (np.reciprocal(a))
```

- **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.

In [51]:

```
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))
```

```
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.

In [53]:

```
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))
```

```
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.

In [55]:

```
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))
```

```
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.]
```

13. NumPy- Statistical Functions

- **numpy.amin()and numpy.amax()**

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

In [62]:

```
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))
```

```
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()**

  function returns the range (maximum-minimum) of values along an axis.

In [3]:

```
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:') #columns
print (np.ptp(a, axis=1))
print ('\n')
print ('Applying ptp() function along axis 0:') #
print (np.ptp(a, axis=0))
```

```
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)

In [5]:

```
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))
```

```
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.

In [7]:

```
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.

In [9]:

```
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()**

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.

In [11]:

```
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))
```

```
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)
```

- **Standard deviation**

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))

In [14]:

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

```
1.118033988749895
```

- **Variance**

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.

In [16]:

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

```
1.25
```

14.NumPy-Sort,Search and counting functions

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

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

numpy.sort()

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

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

Where,

a	Array to be sorted
axis	The axis along which the array is to be sorted. If none, the array is flattened, sorting on the last axis
kind	Default is quicksort
order	If the array contains fields, the order of fields to be sorted

In [18]:

```
a = np.array([[3,7],[9,1]])
print ('Our array is:')
print (a)
print ('\n')
print ('Applying sort() function:')
print (np.sort(a))
print ('\n')
print ('Sort along axis 0:')
print (np.sort(a, axis=0))
print ('\n')
# Order parameter in sort function
dt = np.dtype([('name', 'S10'),('age', int)])
a = np.array([("raju",21),("anil",25),("ravi", 17), ("amar",27)], dtype=dt)
print ('Our array is:')
print (a)
print ('\n')
print ('Order by name:')
print (np.sort(a, order='name'))
```

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


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


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


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


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

numpy.argsort()

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

numpy.lexsort()

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

numpy.argmax() and numpy.argmin()

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

numpy.nonzero()

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

numpy.where()

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

numpy.extract()

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

15.NumPy- Linear algebra

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

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**.

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.

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.

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.

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.

**PANDAS - BASIC

**

- Pandas is an **open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures.**
- Pandas - dervied from the word **Panel Data** - an Econometrics from Multidimensional data.

Key Features of Pandas

1.Fast and efficient DataFrame object with default and customized indexing.

2.Tools for loading data into in-memory data objects from different file formats.

3.Data alignment and integrated handling of missing data.

4.Reshaping and pivoting of date sets.

5.Label-based slicing, indexing and subsetting of large data sets.

6.Columns from a data structure can be deleted or inserted.

7.Group by data for aggregation and transformations.

8.High performance merging and joining of data.

9.Time Series functionality.

- **Pandas deals with the following three data structures –**

1. **Series**
2. **DataFrame**
3. **Panel**

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Mutability

- **All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.**

Note – DataFrame is widely used and one of the most important data structures. Panel is used much less.

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …

Key Points

1. Homogeneous data
2. Size Immutable
3. Values of Data Mutable

DataFrame

- DataFrame is a two-dimensional array with heterogeneous data.
- The data is represented in rows and columns.

Key Points

1. Heterogeneous data
2. Size Mutable
3. Data Mutable

Panel

- Panel is a three-dimensional data structure with heterogeneous data.
- It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

Key Points

1. Heterogeneous data
2. Size Mutable
3. Data Mutable

SERIES

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

pandas.Series(data, index, dtype, copy)

> data

data takes various forms like ndarray, list, constants

> index

Index values must be unique and hashable, same length as data. Default np.arrange(n) if no index is passed.

> dtype

dtype is for data type. If None, data type will be inferred.

> copy

Copy data. Default False

Create an Empty Series

In [ ]:

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print (s)
```

```
Series([], dtype: float64)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3: DeprecationWarning: The d
efault dtype for empty Series will be 'object' instead of 'float64' in a future version.
Specify a dtype explicitly to silence this warning.
  This is separate from the ipykernel package so we can avoid doing imports until
```

Create a Series from ndarray

In [ ]:

```
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print (s)
```

```
0    a
1    b
2    c
3    d
dtype: object
```

In [ ]:

```
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
print (s)
```

```
100    a
101    b
102    c
103    d
dtype: object
```

Create a Series from dict

In [ ]:

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print (s)
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

In [ ]:

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print (s)
```

```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Create a Series from Scalar

In [ ]:

```
s = pd.Series(5, index=[0, 1, 2, 3])
print (s)
```

```
0    5
1    5
2    5
3    5
dtype: int64
```

Accessing Data from Series with Position

In [ ]:

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element
print(s)
print (s[0])
```

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
1
```

In [ ]:

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print (s[:3])
```

```
a    1
b    2
c    3
dtype: int64
```

In [ ]:

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the last three element
print (s[-3:])
```

```
c    3
d    4
e    5
dtype: int64
```

Retrieve Data Using Label (Index)

In [ ]:

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element
print (s['a'])
```

```
1
```

In [ ]:

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
```

```
#retrieve multiple elements
print (s[['a','c','d']])
```

```
a    1
c    3
d    4
dtype: int64
```

If a label is not contained, an exception is raised.

In [ ]:

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements
print (s['f'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.7/dist-packages/pandas/core/indexes/base.py in get_loc(self, key, method, tolerance)
   2897             try:
-> 2898                 return self._engine.get_loc(casted_key)
   2899             except KeyError as err:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'f'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-17-5afc8eba237a> in <module>()
      2 
      3 #retrieve multiple elements
----> 4 print (s['f'])

/usr/local/lib/python3.7/dist-packages/pandas/core/series.py in __getitem__(self, key)
    880 
    881         elif key_is_scalar:
--> 882             return self._get_value(key)
    883 
    884         if is_hashable(key):

/usr/local/lib/python3.7/dist-packages/pandas/core/series.py in _get_value(self, label, takeable)
    988 
    989         # Similar to Index.get_value, but we do not fall back to positional
--> 990         loc = self.index.get_loc(label)
    991         return self.index._get_values_for_loc(self, loc, label)
    992 

/usr/local/lib/python3.7/dist-packages/pandas/core/indexes/base.py in get_loc(self, key, method, tolerance)
   2898                 return self._engine.get_loc(casted_key)
   2899             except KeyError as err:
-> 2900                 raise KeyError(key) from err
   2901 
   2902         if tolerance is not None:

KeyError: 'f'
```

DATAFRAME

- Data is aligned in a tabular fashion in rows and columns.
- A pandas DataFrame can be created using the following constructor –

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

> **data**

data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.

> **index**

For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.

> **columns**

For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed.

> **dtype**

Data type of each column.

> **copy**

This command (or whatever it is) is used for copying of data, if the default is False.

Create an Empty DataFrame

In [ ]:

```
df = pd.DataFrame()
print (df)
```

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

In [ ]:

```
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print (df)
```

```
   0
0  1
1  2
2  3
3  4
4  5
```

In [ ]:

```
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],index=[101,102,103])
print (df)
```

```
       Name   Age
101    Alex   10
```

```
102     Bob   12
103  Clarke   13
```

In [ ]:

```
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],index=[101,102,103],dtype=float)
print (df)
```

```
       Name   Age
101    Alex  10.0
102     Bob  12.0
103  Clarke  13.0
```

Create a DataFrame from Dict of ndarrays / Lists

In [ ]:

```
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print (df)
```

```
    Name  Age
0    Tom   28
1   Jack   34
2  Steve   29
3  Ricky   42
```

In [ ]:

```
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print (df)
```

```
        Name  Age
rank1    Tom   28
rank2   Jack   34
rank3  Steve   29
rank4  Ricky   42
```

Create a DataFrame from List of Dicts

In [ ]:

```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print (df)
```

```
   a   b     c
0  1   2   NaN
1  5  10  20.0
```

In [ ]:

```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print (df)
```

```
        a   b     c
first   1   2   NaN
second  5  10  20.0
```

In [ ]:

```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])
```

```
#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print (df1)
print (df2)
```

```
        a   b
first   1   2
second  5  10
        a  b1
first   1 NaN
second  5 NaN
```

Create a DataFrame from Dict of Series

In [ ]:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print (df)
```

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
```

Column Selection

In [ ]:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
print(df['one'])
```

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

Column Addition

In [ ]:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

# Adding a new column to an existing DataFrame object with column label by passing new se
ries

print ("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],index=['a','b','c'])
print (df)
```

```
print ("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']

print (df)
```

```
Adding a new column by passing as Series:
   one  two  three
a  1.0    1   10.0
b  2.0    2   20.0
c  3.0    3   30.0
d  NaN    4    NaN
Adding a new column using the existing columns in DataFrame:
   one  two  three  four
a  1.0    1   10.0  11.0
b  2.0    2   20.0  22.0
c  3.0    3   30.0  33.0
d  NaN    4    NaN   NaN
```

Column Deletion

In [ ]:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
   'three' : pd.Series([10,20,30], index=['a','b','c'])}

df = pd.DataFrame(d)
print ("Our dataframe is:")
print (df)

# using del function
print ("Deleting the first column using DEL function:")
del (df['one'])
print (df)

# using pop function
print ("Deleting another column using POP function:")
df.pop('two')
print (df)
```

```
Our dataframe is:
   one  two  three
a  1.0    1   10.0
b  2.0    2   20.0
c  3.0    3   30.0
d  NaN    4    NaN
Deleting the first column using DEL function:
   two  three
a    1   10.0
b    2   20.0
c    3   30.0
d    4    NaN
Deleting another column using POP function:
   three
a   10.0
b   20.0
c   30.0
d    NaN
```

Row Selection, Addition, and Deletion

Selection by Label

Rows can be selected by passing row label to a loc function.

In [ ]:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
print (df.loc['b'])
```

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
one    2.0
two    2.0
Name: b, dtype: float64
```

Selection by integer location

Rows can be selected by passing integer location to an iloc function.

In [ ]:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
print (df.iloc[2])
```

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
one    3.0
two    3.0
Name: c, dtype: float64
```

Slice Rows

Multiple rows can be selected using ' : ' operator.

In [ ]:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
print (df[2:4])
```

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
   one  two
c  3.0    3
d  NaN    4
```

Addition of Rows

Add new rows to a DataFrame using the append function. This function will append the rows at the end.

In [ ]:

```
df = pd.DataFrame([[1, 2, 5], [3, 4, 8]], columns = ['a','b','c'])
df2 = pd.DataFrame([[5, 6, 3], [7, 8, 4]], columns = ['a','b','c'])
```

```
df = df.append(df2)
print(df)
```

```
   a  b  c
0  1  2  5
1  3  4  8
0  5  6  3
1  7  8  4
```

Deletion of Rows

In [ ]:

```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0
df = df.drop(0)
del df['a'] #deleting the column
print (df)
```

```
   b
1  4
1  8
```

PANEL

- **A panel is a 3D container of data.**
- **The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data. They are –**
  1. **items – axis 0, each item corresponds to a DataFrame contained inside.**
  2. **major_axis – axis 1, it is the index (rows) of each of the DataFrames.**
  3. **minor_axis – axis 2, it is the columns of each of the DataFrames.**

A Panel can be created using the following constructor –

```
pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)
```

> **data: Data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame**
>
> **items: axis=0**
>
> **major_axis: axis=1**
>
> **minor_axis: axis=2**
>
> **dtype: Data type of each column**
>
> **copy: Copy data. Default, false**

Basic Functionality

Series Basic Functionality

Sr.No	Attribute or Method & Description

Sr.No.	Attribute or Method & Description
1	**axes** Returns a list of the row axis labels
2	**dtype** Returns the dtype of the object.
3	**empty** Returns True if series is empty.
4	**ndim** Returns the number of dimensions of the underlying data, by definition 1.
5	**size** Returns the number of elements in the underlying data.
6	**values** Returns the Series as ndarray.
7	**head()** Returns the first n rows.
8	**tail()** Returns the last n rows.

In [ ]:

```
s = pd.Series(np.random.randn(4))
print (s)
```

```
0    0.343856
1   -1.513000
2    1.177414
3    1.082206
dtype: float64
```

axes

Returns the list of the labels of the series.

In [ ]:

```
#Create a series with 100 random numbers
s = pd.Series(np.random.randn(4))
print ("The axes are:")
print (s.axes)
```

```
The axes are:
[RangeIndex(start=0, stop=4, step=1)]
```

empty

Returns the Boolean value saying whether the Object is empty or not. True indicates that the object is empty.

In [ ]:

```
#Create a series with 100 random numbers
s = pd.Series(np.random.randn(4))
print ("Is the Object empty?")
print (s.empty)
```

```
Is the Object empty?
False
```

ndim

Returns the number of dimensions of the object. By definition, a Series is a 1D data structure, so it returns

In [ ]:

```
#Create a series with 4 random numbers
s = pd.Series(np.random.randn(4))
print (s)

print ("The dimensions of the object:")
print (s.ndim)
```

```
0    1.917447
1   -0.802830
2    1.723864
3    1.865692
dtype: float64
The dimensions of the object:
1
```

size

Returns the size(length) of the series.

In [ ]:

```
#Create a series with 4 random numbers
s = pd.Series(np.random.randn(2))
print (s)
print ("The size of the object:")
print (s.size)
```

```
0    1.18674
1    1.21442
dtype: float64
The size of the object:
2
```

values

Returns the actual data in the series as an array.

In [ ]:

```
#Create a series with 4 random numbers
s = pd.Series(np.random.randn(4))
print (s)

print ("The actual data series is:")
print (s.values)
```

```
0    0.939080
1   -0.149085
2   -1.030239
3   -1.327734
dtype: float64
```

```
The actual data series is:
[ 0.93907996 -0.1490846  -1.03023938 -1.3277343 ]
```

Head & Tail

head() returns the first n rows(observe the index values). The default number of elements to display is five, but you may pass a custom number.

tail() returns the last n rows(observe the index values). The default number of elements to display is five, but you may pass a custom number.

In [ ]:

```
#Create a series with 4 random numbers
s = pd.Series(np.random.randn(4))
print ("The original series is:")
print (s)

print ("The first two rows of the data series:")
print (s.head(2))
```

```
The original series is:
0    0.066872
1   -0.448631
2    1.367534
3    1.014159
dtype: float64
The first two rows of the data series:
0    0.066872
1   -0.448631
dtype: float64
```

In [ ]:

```
#Create a series with 4 random numbers
s = pd.Series(np.random.randn(4))
print ("The original series is:")
print (s)

print ("The last two rows of the data series:")
print (s.tail(2))
```

```
The original series is:
0    0.857251
1    1.228845
2    2.350523
3   -1.389315
dtype: float64
The last two rows of the data series:
2    2.350523
3   -1.389315
dtype: float64
```

DataFrame Basic Functionality

Sr.No.	Attribute or Method & Description
1	**T** Transposes rows and columns.
2	**axes** Returns a list with the row axis labels and column axis labels as the only members.
3	**dtypes** Returns the dtypes in this object.

4	**empty** True if NDFrame is entirely empty [no items]; if any of the axes are of length 0.
5	**ndim** Number of axes / array dimensions.
6	**shape** Returns a tuple representing the dimensionality of the DataFrame.
7	**size** Number of elements in the NDFrame.
8	**values** Numpy representation of NDFrame.
9	**head()** Returns the first n rows.
10	**tail()** Returns last n rows.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

#Create a DataFrame
df = pd.DataFrame(d)
print ("Our data series is:")
print (df)
```

```
Our data series is:
    Name  Age  Rating
0    Tom   25    4.23
1  James   26    3.24
2  Ricky   25    3.98
3    Vin   23    2.56
4  Steve   30    3.20
5  Smith   29    4.60
6   Jack   23    3.80
```

T (Transpose)

Returns the transpose of the DataFrame. The rows and columns will interchange.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

# Create a DataFrame
df = pd.DataFrame(d)
print ("The transpose of the data series is:")
print (df.T)
```

```
The transpose of the data series is:
           0      1      2     3      4      5     6
```

```
Name      Tom  James  Ricky   Vin  Steve  Smith  Jack
Age        25     26     25    23     30     29    23
Rating   4.23   3.24   3.98  2.56    3.2    4.6   3.8
```

axes

Returns the list of row axis labels and column axis labels.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

#Create a DataFrame
df = pd.DataFrame(d)
print ("Row axis labels and column axis labels are:")
print (df.axes)
```

```
Row axis labels and column axis labels are:
[RangeIndex(start=0, stop=7, step=1), Index(['Name', 'Age', 'Rating'], dtype='object')]
```

dtypes

Returns the data type of each column.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

#Create a DataFrame
df = pd.DataFrame(d)
print ("The data types of each column are:")
print (df.dtypes)
```

```
The data types of each column are:
Name       object
Age         int64
Rating    float64
dtype: object
```

empty

Returns the Boolean value saying whether the Object is empty or not; True indicates that the object is empty.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

#Create a DataFrame
df = pd.DataFrame(d)
print ("Is the object empty?")
print (df.empty)
```

```
Is the object empty?
False
```

ndim

Returns the number of dimensions of the object. By definition, DataFrame is a 2D object.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
```

```
    'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

#Create a DataFrame
df = pd.DataFrame(d)
print ("Our object is:")
print (df)
print ("The dimension of the object is:")
print (df.ndim)
```

```
Our object is:
    Name  Age  Rating
0    Tom   25    4.23
1  James   26    3.24
2  Ricky   25    3.98
3    Vin   23    2.56
4  Steve   30    3.20
5  Smith   29    4.60
6   Jack   23    3.80
The dimension of the object is:
2
```

shape

Returns a tuple representing the dimensionality of the DataFrame. Tuple (a,b), where a represents the number of rows and b represents the number of columns.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
    'Age':pd.Series([25,26,25,23,30,29,23]),
    'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

#Create a DataFrame
df = pd.DataFrame(d)
print ("Our object is:")
print (df)
print ("The shape of the object is:")
print (df.shape)
print(df.size)
print(df.values)
```

```
Our object is:
    Name  Age  Rating
0    Tom   25    4.23
1  James   26    3.24
2  Ricky   25    3.98
3    Vin   23    2.56
4  Steve   30    3.20
5  Smith   29    4.60
6   Jack   23    3.80
The shape of the object is:
(7, 3)
21
[['Tom' 25 4.23]
 ['James' 26 3.24]
 ['Ricky' 25 3.98]
 ['Vin' 23 2.56]
 ['Steve' 30 3.2]
 ['Smith' 29 4.6]
 ['Jack' 23 3.8]]
```

Descriptive Statistics

Sr.No.	Function	Description
1	count()	Number of non-null observations

2	sum()	Sum of values
3	mean()	Mean of Values
4	median()	Median of Values
5	mode()	Mode of values
6	std()	Standard Deviation of the Values
7	min()	Minimum Value
8	max()	Maximum Value
9	abs()	Absolute Value
10	prod()	Product of Values
11	cumsum()	Cumulative Sum
12	cumprod()	Cumulative Product

Note – Since DataFrame is a Heterogeneous data structure. Generic operations don't work with all functions.

- **Functions like sum(), cumsum() work with both numeric and character (or) string data elements without any error. Though n practice, character aggregations are never used generally, these functions do not throw any exception.**
- **Functions like abs(), cumprod() throw exception when the DataFrame contains character or string data because such operations cannot be performed.**

Summarizing Data

The describe() function computes a summary of statistics pertaining to the DataFrame columns.

In [ ]:

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print (df.describe())
print (df.describe(include=['object']))
print (df. describe(include='all'))
```

```
              Age      Rating
count   12.000000   12.000000
mean    31.833333    3.743333
std      9.232682    0.661628
min     23.000000    2.560000
25%     25.000000    3.230000
50%     29.500000    3.790000
75%     35.500000    4.132500
max     51.000000    4.800000
          Name
count       12
unique      12
top     Andres
```

```
freq          1
           Name        Age     Rating
count        12  12.000000  12.000000
unique       12        NaN        NaN
top     Andres        NaN        NaN
freq          1        NaN        NaN
mean        NaN  31.833333   3.743333
std         NaN   9.232682   0.661628
min         NaN  23.000000   2.560000
25%         NaN  25.000000   3.230000
50%         NaN  29.500000   3.790000
75%         NaN  35.500000   4.132500
max         NaN  51.000000   4.800000
```

Function Application

- **Table wise Function Application: pipe()**
- **Row or Column Wise Function Application: apply()**
- **Element wise Function Application: applymap()**

Table-wise Function Application

In [ ]:

```
def adder(ele1,ele2):
   return ele1+ele2

df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
print(df)
df.pipe(adder,2)
print (df.apply(np.mean))
```

```
       col1      col2      col3
0  0.231941  0.816780 -0.807209
1  0.201243  0.170274  0.118749
2  0.657235  0.738078  0.238683
3 -0.749622  0.108821  0.557177
4  0.411946  0.834896  1.015541
col1    0.150548
col2    0.533770
col3    0.224588
dtype: float64
```

Row or Column Wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame or Panel using the apply() method, which, like the descriptive statistics methods, takes an optional axis argument. By default, the operation performs column wise, taking each column as an array-like.

In [ ]:

```
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.apply(np.mean)
print (df.apply(np.mean))
```

```
col1   -0.155697
col2    0.140364
col3   -0.481523
dtype: float64
```

In [ ]:

```
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.apply(np.mean,axis=1)
print (df.apply(np.mean))
```

```
col1     0.198752
col2    -0.003511
col3    -0.336391
dtype: float64
```

In [ ]:

```
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.apply(lambda x: x.max() - x.min())
print (df.apply(np.mean))
```

```
col1    -0.016829
col2     1.181789
col3     0.646476
dtype: float64
```

Element Wise Function Application

Not all functions can be vectorized (neither the NumPy arrays which return another array nor any value), the methods applymap() on DataFrame and analogously map() on Series accept any Python function taking a single value and returning a single value.

In [ ]:

```
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])

# My custom function
df['col1'].map(lambda x:x*100)
print (df.apply(np.mean))
```

```
col1    -0.015860
col2     0.464819
col3     0.316629
dtype: float64
```

In [ ]:

```
# My custom function
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.applymap(lambda x:x*100)
print (df.apply(np.mean))
```

```
col1     0.299843
col2     0.373885
col3     0.440734
dtype: float64
```

Reindexing

- **changes the row labels and column labels of a DataFrame.**
- **Multiple operations can be accomplished through indexing like –**

```
Reorder the existing data to match a new set of labels.

Insert missing value (NA) markers in label locations where no data for the labe
l existed.
```

In [ ]:

```
N=20

df = pd.DataFrame({
   'A': pd.date_range(start='2016-01-01',periods=N,freq='D'),
   'x': np.linspace(0,stop=N-1,num=N),
   'y': np.random.rand(N),
```

```
    'C': np.random.choice(['Low','Medium','High'],N).tolist(),
    'D': np.random.normal(100, 10, size=(N)).tolist()
})
print(df)
#reindex the DataFrame
df_reindexed = df.reindex(index=[0,2,5], columns=['A', 'C', 'B'])

print (df_reindexed)
```

```
            A     x         y       C           D
0  2016-01-01   0.0  0.976645  Medium  104.517588
1  2016-01-02   1.0  0.240366    High  121.305194
2  2016-01-03   2.0  0.264866     Low   85.483048
3  2016-01-04   3.0  0.308523  Medium  101.167788
4  2016-01-05   4.0  0.660320    High   83.188331
5  2016-01-06   5.0  0.722134     Low   96.122472
6  2016-01-07   6.0  0.628637    High   71.064031
7  2016-01-08   7.0  0.056529     Low  106.281955
8  2016-01-09   8.0  0.489087    High  120.610635
9  2016-01-10   9.0  0.579790  Medium  105.702452
10 2016-01-11  10.0  0.071238    High  109.295007
11 2016-01-12  11.0  0.090723    High   95.717132
12 2016-01-13  12.0  0.429125     Low   93.577057
13 2016-01-14  13.0  0.556517  Medium   95.939707
14 2016-01-15  14.0  0.455765  Medium  114.938609
15 2016-01-16  15.0  0.056575    High  105.417323
16 2016-01-17  16.0  0.995810  Medium   91.307640
17 2016-01-18  17.0  0.488257     Low   99.661433
18 2016-01-19  18.0  0.442453     Low   95.611593
19 2016-01-20  19.0  0.927716    High  111.626369
           A       C   B
0 2016-01-01  Medium NaN
2 2016-01-03     Low NaN
5 2016-01-06     Low NaN
```

Reindex to Align with Other Objects

In [ ]:

```
df1 = pd.DataFrame(np.random.randn(10,3),columns=['col1','col2','col3'])
df2 = pd.DataFrame(np.random.randn(7,3),columns=['col1','col2','col3'])
print(df1)
df1 = df1.reindex_like(df2)
print (df1)
```

```
       col1      col2      col3
0  0.347836  2.676956  0.833656
1 -0.814652 -0.092549 -1.301584
2  0.418241  1.108528 -0.594561
3 -0.889054 -0.732778 -0.051032
4 -0.733484 -1.890372 -0.549061
5  1.817873 -1.389579  0.095550
6 -1.230041  0.083473 -0.533115
7 -1.558132  0.061127  0.222137
8  0.522397 -0.013771 -0.836590
9  0.583338  1.030694 -1.256451
       col1      col2      col3
0  0.347836  2.676956  0.833656
1 -0.814652 -0.092549 -1.301584
2  0.418241  1.108528 -0.594561
3 -0.889054 -0.732778 -0.051032
4 -0.733484 -1.890372 -0.549061
5  1.817873 -1.389579  0.095550
6 -1.230041  0.083473 -0.533115
```

Note – Here, the df1 DataFrame is altered and reindexed like df2. The column names should be matched or else NAN will be added for the entire column label.

Filling while ReIndexing

Filling while Reindexing

reindex() takes an optional parameter method which is a filling method with values as follows –

pad/ffill – Fill values forward

bfill/backfill – Fill values backward

nearest – Fill from the nearest index values

In [ ]:

```
df1 = pd.DataFrame(np.random.randn(6,3),columns=['col1','col2','col3'])
df2 = pd.DataFrame(np.random.randn(2,3),columns=['col1','col2','col3'])

# Padding NAN's
print (df2.reindex_like(df1))

# Now Fill the NAN's with preceding Values
print ("Data Frame with Forward Fill:")
print (df2.reindex_like(df1,method='ffill'))
```

```
       col1      col2      col3
0 -0.460393 -1.107607 -1.058472
1  1.903622 -1.506541  1.467214
2       NaN       NaN       NaN
3       NaN       NaN       NaN
4       NaN       NaN       NaN
5       NaN       NaN       NaN
Data Frame with Forward Fill:
       col1      col2      col3
0 -0.460393 -1.107607 -1.058472
1  1.903622 -1.506541  1.467214
2  1.903622 -1.506541  1.467214
3  1.903622 -1.506541  1.467214
4  1.903622 -1.506541  1.467214
5  1.903622 -1.506541  1.467214
```

Limits on Filling while Reindexing

The limit argument provides additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches.

In [ ]:

```
df1 = pd.DataFrame(np.random.randn(6,3),columns=['col1','col2','col3'])
df2 = pd.DataFrame(np.random.randn(2,3),columns=['col1','col2','col3'])

# Padding NAN's
print (df2.reindex_like(df1))

# Now Fill the NAN's with preceding Values
print ("Data Frame with Forward Fill limiting to 1:")
print (df2.reindex_like(df1,method='ffill',limit=1))
```

```
       col1      col2      col3
0 -0.811587 -0.293629  1.603325
1 -0.451165 -0.868913 -0.511139
2       NaN       NaN       NaN
3       NaN       NaN       NaN
4       NaN       NaN       NaN
5       NaN       NaN       NaN
Data Frame with Forward Fill limiting to 1:
       col1      col2      col3
0 -0.811587 -0.293629  1.603325
1 -0.451165 -0.868913 -0.511139
2 -0.451165 -0.868913 -0.511139
3       NaN       NaN       NaN
4       NaN       NaN       NaN
5       NaN       NaN       NaN
```

```
5        NaN        NaN        NaN
```

Renaming

The rename() method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

In [ ]:

```
df1 = pd.DataFrame(np.random.randn(6,3),columns=['col1','col2','col3'])
print (df1)

print ("After renaming the rows and columns:")
print (df1.rename(columns={'col1' : 'c1', 'col2' : 'c2'},index = {0 : 'apple', 1 : 'bana
na', 2 : 'durian'}))
```

```
       col1      col2      col3
0  0.906990 -1.412007 -0.185552
1 -0.188860 -0.461197  0.030534
2  0.513972  1.462018  1.203715
3 -2.139031  0.638142 -1.218196
4  1.533187  0.780998  1.783094
5 -1.543924 -0.604516  1.416142
After renaming the rows and columns:
              c1        c2      col3
apple   0.906990 -1.412007 -0.185552
banana -0.188860 -0.461197  0.030534
durian  0.513972  1.462018  1.203715
3      -2.139031  0.638142 -1.218196
4       1.533187  0.780998  1.783094
5      -1.543924 -0.604516  1.416142
```

Iteration

Iterating a DataFrame

In [ ]:

```
import pandas as pd
import numpy as np
N=20
df = pd.DataFrame({
   'A': pd.date_range(start='2016-01-01',periods=N,freq='D'),
   'x': np.linspace(0,stop=N-1,num=N),
   'y': np.random.rand(N),
   'C': np.random.choice(['Low','Medium','High'],N).tolist(),
   'D': np.random.normal(100, 10, size=(N)).tolist()
   })
print(df)

for col in df:
   print (col) #prints only the column names
```

```
            A     x         y       C           D
0  2016-01-01   0.0  0.035779    High  111.817182
1  2016-01-02   1.0  0.733338     Low   96.778778
2  2016-01-03   2.0  0.622492    High   98.068822
3  2016-01-04   3.0  0.124783     Low   84.848860
4  2016-01-05   4.0  0.903839    High   97.794692
5  2016-01-06   5.0  0.704438     Low   92.127049
6  2016-01-07   6.0  0.541755     Low   89.904608
7  2016-01-08   7.0  0.808136     Low   86.437755
8  2016-01-09   8.0  0.523136     Low   92.393352
9  2016-01-10   9.0  0.041684     Low   95.816143
10 2016-01-11  10.0  0.431267    High   98.979742
11 2016-01-12  11.0  0.703921     Low   95.371405
12 2016-01-13  12.0  0.885836     Low   88.057842
13 2016-01-14  13.0  0.193648  Medium   99.277544
```

```
14 2016-01-15  14.0  0.697265    High   97.032961
15 2016-01-16  15.0  0.019812    High   85.819187
16 2016-01-17  16.0  0.643544     Low  112.747346
17 2016-01-18  17.0  0.163281  Medium   93.449913
18 2016-01-19  18.0  0.230668     Low   98.316041
19 2016-01-20  19.0  0.438507     Low   99.943363
A
x
y
C
D
```

To iterate over the rows of the DataFrame, we can use the following functions –

- **iteritems() – to iterate over the (key,value) pairs**
- **iterrows() – iterate over the rows as (index,series) pairs**
- **itertuples() – iterate over the rows as namedtuples**

iteritems()

In [ ]:

```
df = pd.DataFrame(np.random.randn(4,3),columns=['col1','col2','col3'])
print(df)
for key,value in df.iteritems():
   print (key,value)
```

```
       col1      col2      col3
0  1.168326  1.505951  0.346934
1  0.720975  1.667640  1.182600
2 -0.311300  0.593203  0.677633
3 -0.618822 -0.463293 -0.242687
col1 0    1.168326
1    0.720975
2   -0.311300
3   -0.618822
Name: col1, dtype: float64
col2 0    1.505951
1    1.667640
2    0.593203
3   -0.463293
Name: col2, dtype: float64
col3 0    0.346934
1    1.182600
2    0.677633
3   -0.242687
Name: col3, dtype: float64
```

each column is iterated separately as a key-value pair in a Series.

iterrows()

In [ ]:

```
df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])
print(df)
for row_index,row in df.iterrows():
   print (row_index,row)
```

```
       col1      col2      col3
0  1.444090  0.046375 -0.848445
1  1.836205  1.560163  0.122909
2 -1.271620  0.347389 -0.713816
3 -0.484960  1.742473 -0.881741
0 col1    1.444090
col2    0.046375
col3   -0.848445
Name: 0, dtype: float64
```

```
Name: 0, dtype: float64
1 col1    1.836205
col2    1.560163
col3    0.122909
Name: 1, dtype: float64
2 col1   -1.271620
col2    0.347389
col3   -0.713816
Name: 2, dtype: float64
3 col1   -0.484960
col2    1.742473
col3   -0.881741
Name: 3, dtype: float64
```

Note – Because iterrows() iterate over the rows, it doesn't preserve the data type across the row. 0,1,2 are the row indices and col1,col2,col3 are column indices

itertuples()

In [ ]:

```
df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])
print(df)
for row in df.itertuples():
    print (row)
```

```
       col1      col2      col3
0 -2.077645 -1.731257  0.064170
1 -0.748485 -0.708296 -0.846344
2  1.404543 -0.514144 -0.072735
3  0.107781 -0.514891 -0.695101
Pandas(Index=0, col1=-2.077644507376731, col2=-1.7312565192617564, col3=0.064170082261585
81)
Pandas(Index=1, col1=-0.7484849511665813, col2=-0.708296270930595, col3=-0.84634377688326
47)
Pandas(Index=2, col1=1.4045428299067606, col2=-0.5141439880110376, col3=-0.07273473255283
526)
Pandas(Index=3, col1=0.10778083145939131, col2=-0.5148912143446676, col3=-0.6951014016455
861)
```

Note – Do not try to modify any object while iterating. Iterating is meant for reading and the iterator returns a copy of the original object (a view), thus the changes will not reflect on the original object.

In [ ]:

```
df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])
print(df)
for index, row in df.iterrows():
   row['a'] = 10
print (df)
```

```
       col1      col2      col3
0 -0.457023  1.008755 -2.590143
1 -0.203070  0.474942 -0.213065
2 -1.377836 -0.466549  1.784151
3  0.914468  0.449972 -1.747829
       col1      col2      col3
0 -0.457023  1.008755 -2.590143
1 -0.203070  0.474942 -0.213065
2 -1.377836 -0.466549  1.784151
3  0.914468  0.449972 -1.747829
```

Observe, no changes reflected.

Sorting

- By label

- By label
- By Actual Value

In [ ]:

```
unsorted_df=pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],columns=['col
2','col1'])
print (unsorted_df)
```

```
       col2      col1
1  0.130174 -0.262908
4 -1.097250 -1.017719
6 -0.509985  2.439330
2 -1.068080 -0.236300
3  0.351354  0.219491
5  1.489364 -0.148715
9 -0.370576  0.183798
8 -0.565695 -1.296003
0 -0.503785 -0.722336
7  0.532778 -3.187010
```

In unsorted_df, the labels and the values are unsorted.

By Label

By default, sorting is done on row labels in ascending order.

In [ ]:

```
unsorted_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],columns = [
'col2','col1'])
sorted_df=unsorted_df.sort_index()
print (sorted_df)
```

```
       col2      col1
0  1.054777  0.253568
1 -1.690379 -1.335536
2 -0.115155 -0.768089
3 -0.313425  0.098405
4 -0.427349 -0.774873
5 -0.885698  0.258032
6  0.139761  0.365469
7 -1.393712 -0.935471
8  0.361366 -1.689609
9  0.869936  0.381890
```

Order of Sorting

In [ ]:

```
unsorted_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],columns = [
'col2','col1'])

sorted_df = unsorted_df.sort_index(ascending=False)
print (sorted_df)
```

```
       col2      col1
9 -1.050314  0.953408
8  1.049372 -0.999049
7  0.578397 -0.940296
6 -0.889694 -0.125085
5 -0.060960 -0.032357
4 -0.069340 -0.300917
3 -1.756568  0.930280
2  0.107822  1.560969
1 -1.164749 -1.974579
0 -0.265541  0.544790
```

Sort the Columns

In [ ]:

```
unsorted_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],columns = [
'col2','col1'])

sorted_df=unsorted_df.sort_index(axis=1)

print (sorted_df)
```

```
       col1      col2
1 -0.259607  0.104600
4  0.297673 -1.564816
6 -1.384420  0.219156
2 -0.142928 -1.431338
3 -1.308655  2.326203
5  0.392084  2.712869
9  1.130300 -0.625928
8 -1.130491 -0.693736
0 -0.755032  1.472285
7 -1.491992  0.206592
```

By Value

In [ ]:

```
unsorted_df = pd.DataFrame({'col1':[2,1,1,1],'col2':[1,3,2,4]})
sorted_df = unsorted_df.sort_values(by='col1')
print(sorted_df)
```

```
   col1  col2
1     1     3
2     1     2
3     1     4
0     2     1
```

'by' argument takes a list of column values.

In [ ]:

```
unsorted_df = pd.DataFrame({'col1':[2,1,1,1],'col2':[1,3,2,4]})
sorted_df = unsorted_df.sort_values(by=['col1','col2'])

print (sorted_df)
```

```
   col1  col2
2     1     2
1     1     3
3     1     4
0     2     1
```

Sorting Algorithm

sort_values() provides a provision to choose the algorithm from mergesort, heapsort and quicksort. Mergesort is the only stable algorithm.

In [ ]:

```
unsorted_df = pd.DataFrame({'col1':[2,1,1,1],'col2':[1,3,2,4]})
sorted_df = unsorted_df.sort_values(by='col1' ,kind='mergesort')
print (sorted_df)
```

```
   col1  col2
1     1     3
2     1     2
```

```
3    1    4
0    2    1
```

Working with Text Data

Sr.No	Function & Description
1	**lower()** Converts strings in the Series/Index to lower case.
2	**upper()** Converts strings in the Series/Index to upper case.
3	**len()** Computes String length().
4	**strip()** Helps strip whitespace(including newline) from each string in the Series/index from both the sides.
5	**split(' ')** Splits each string with the given pattern.
6	**cat(sep=' ')** Concatenates the series/index elements with given separator.
7	**get_dummies()** Returns the DataFrame with One-Hot Encoded values.
8	**contains(pattern)** Returns a Boolean value True for each element if the substring contains in the element, else False.
9	**replace(a,b)** Replaces the value **a** with the value **b**.
10	**repeat(value)** Repeats each element with specified number of times.
11	**count(pattern)** Returns count of appearance of pattern in each element.
12	**startswith(pattern)** Returns true if the element in the Series/Index starts with the pattern.
13	**endswith(pattern)** Returns true if the element in the Series/Index ends with the pattern.

14	**find(pattern)** Returns the first position of the first occurrence of the pattern.
15	**findall(pattern)** Returns a list of all occurrence of the pattern.
16	**swapcase** Swaps the case lower/upper.
17	**islower()** Checks whether all characters in each string in the Series/Index in lower case or not. Returns Boolean
18	**isupper()** Checks whether all characters in each string in the Series/Index in upper case or not. Returns Boolean.
19	**isnumeric()** Checks whether all characters in each string in the Series/Index are numeric. Returns Boolean.

Options and Customization

1. **get_option()**
2. **set_option()**
3. **reset_option()**
4. **describe_option()**
5. **option_context()**

display.max_rows

Displays the default number of value. Interpreter reads this value and displays the rows with this value as upper limit to display.

In [ ]:

```
import pandas as pd
print (pd.get_option("display.max_rows"))
```

```
60
```

display.max_columns

Displays the default number of value. Interpreter reads this value and displays the rows with this value as upper limit to display.

In [ ]:

```
import pandas as pd
import numpy as np
print (pd.get_option("display.max_columns"))
```

```
0
```

Here, 60 and 20 are the default configuration parameter values.

Indexing and Selecting Data

Sr.No	Indexing & Description
1	**.loc()** Label based
2	**.iloc()** Integer based
3	**.ix()** Both Label and Integer based

.loc()

- **Pandas provide various methods to have purely label based indexing.**
- **loc takes two single/list/range operator separated by ','. The first one indicates the row and the second one indicates columns.**

In [ ]:

```
df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B', 'C', 'D'])
print(df)
#select all rows for a specific column
print (df.loc[:,'A'])
print (df.loc[:,['A','C']])
print (df.loc[['a','b','f','h'],['A','C']])
print (df.loc['a':'h'])
print (df.loc['a']>0)
```

```
          A         B         C         D
a  1.812027 -1.779037  0.226790  0.617836
b  1.303573  0.247269 -1.380891 -1.060415
c -0.093389  0.191240  0.613098  0.837150
d  0.093750  0.052477  1.865056  0.279955
e -0.161892  0.266479 -1.340509 -2.402076
f  1.450524  0.425768  0.731206 -0.140212
g -0.337239  1.139402  0.338786  0.782365
h  0.224944 -1.659678  1.519790  1.262690
a    1.812027
b    1.303573
c   -0.093389
d    0.093750
e   -0.161892
f    1.450524
g   -0.337239
h    0.224944
Name: A, dtype: float64
          A         C
a  1.812027  0.226790
b  1.303573 -1.380891
c -0.093389  0.613098
```

```
d  0.093750  1.865056
e -0.161892 -1.340509
f  1.450524  0.731206
g -0.337239  0.338786
h  0.224944  1.519790
          A         C
a  1.812027  0.226790
b  1.303573 -1.380891
f  1.450524  0.731206
h  0.224944  1.519790
          A         B         C         D
a  1.812027 -1.779037  0.226790  0.617836
b  1.303573  0.247269 -1.380891 -1.060415
c -0.093389  0.191240  0.613098  0.837150
d  0.093750  0.052477  1.865056  0.279955
e -0.161892  0.266479 -1.340509 -2.402076
f  1.450524  0.425768  0.731206 -0.140212
g -0.337239  1.139402  0.338786  0.782365
h  0.224944 -1.659678  1.519790  1.262690
A     True
B    False
C     True
D     True
Name: a, dtype: bool
```

.iloc()

Pandas provide various methods in order to get purely integer based indexing.

In [ ]:

```python
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])

# select all rows for a specific column
print (df.iloc[:4])
print (df.iloc[:4])
print (df.iloc[1:5, 2:4])
print (df.iloc[[1, 3, 5], [1, 3]])
print (df.iloc[1:3, :])
print (df.iloc[:,1:3])
```

```
          A         B         C         D
0  0.796507  1.969190  0.005667 -1.243480
1  0.072288  1.007846  0.763091 -0.725836
2 -0.379662 -0.226257 -0.015324  0.185846
3 -0.191053 -0.217608 -1.458507 -0.679456
          A         B         C         D
0  0.796507  1.969190  0.005667 -1.243480
1  0.072288  1.007846  0.763091 -0.725836
2 -0.379662 -0.226257 -0.015324  0.185846
3 -0.191053 -0.217608 -1.458507 -0.679456
          C         D
1  0.763091 -0.725836
2 -0.015324  0.185846
3 -1.458507 -0.679456
4 -1.846577 -1.068272
          B         D
1  1.007846 -0.725836
3 -0.217608 -0.679456
5  1.496485 -0.285443
          A         B         C         D
1  0.072288  1.007846  0.763091 -0.725836
2 -0.379662 -0.226257 -0.015324  0.185846
          B         C
0  1.969190  0.005667
1  1.007846  0.763091
2 -0.226257 -0.015324
3 -0.217608 -1.458507
4  0.805319 -1.846577
5  1.496485 -0.042291
```

```
6 -1.043700  0.484291
7  0.551094  1.059561
```

.ix()

Besides pure label based and integer based, Pandas provides a hybrid method for selections and subsetting the object using the .ix() operator.

Statistical Functions

Percent_change

Series, DatFrames and Panel, all have the function pct_change().

This function compares every element with its prior element and computes the change percentage.

In [ ]:

```
s = pd.Series([1,2,3,4,5,4])
print (s.pct_change())

df = pd.DataFrame(np.random.randn(5, 2))
print (df.pct_change())
```

```
0         NaN
1    1.000000
2    0.500000
3    0.333333
4    0.250000
5   -0.200000
dtype: float64
          0         1
0       NaN       NaN
1  1.618223 -1.559208
2 -1.270688 -4.108429
3 -5.935224 -1.249026
4 -1.822920  1.230661
```

By default, the pct_change() operates on columns; if you want to apply the same row wise, then use axis=1() argument.

Covariance

Covariance is applied on series data. The Series object has a method cov to compute covariance between series objects. NA will be excluded automatically.

In [ ]:

```
s1 = pd.Series(np.random.randn(10))
s2 = pd.Series(np.random.randn(10))
print (s1.cov(s2))
```

```
-0.21952515787455573
```

Window Functions

Pandas provide few variants like rolling, expanding and exponentially moving weights for window statistics.

.rolling() Function

This function can be applied on a series of data. Specify the window=n argument and apply the appropriate

statistical function on top of it.

.expanding() Function

This function can be applied on a series of data. Specify the min_periods=n argument and apply the appropriate statistical function on top of it.

.ewm() Function

ewm is applied on a series of data. Specify any of the com, span, halflife argument and apply the appropriate statistical function on top of it. It assigns the weights exponentially.

In [ ]:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])
print (df.rolling(window=3).mean())
print("\n")
print (df.expanding(min_periods=3).mean())
print("\n")
print (df.ewm(com=0.5).mean())
```

```
                   A         B         C         D
2000-01-01       NaN       NaN       NaN       NaN
2000-01-02       NaN       NaN       NaN       NaN
2000-01-03  0.082538  0.344065 -0.027439 -0.435739
2000-01-04 -0.864256 -0.061896  0.100686 -0.104483
2000-01-05 -1.170429  0.112589  0.307739  0.353499
2000-01-06 -0.954294  0.005968 -0.590292  0.395864
2000-01-07 -0.909734  0.163456 -0.194363 -0.630932
2000-01-08 -0.462125 -0.019599 -0.803848 -1.221888
2000-01-09 -0.565005 -0.221935 -0.191522 -0.763835
2000-01-10  0.043026 -0.567516  0.301674 -0.387649


                   A         B         C         D
2000-01-01       NaN       NaN       NaN       NaN
2000-01-02       NaN       NaN       NaN       NaN
2000-01-03  0.082538  0.344065 -0.027439 -0.435739
2000-01-04 -0.296779  0.063343 -0.128987 -0.054292
2000-01-05 -0.303398  0.265960 -0.059773  0.081088
2000-01-06 -0.435878  0.175017 -0.308866 -0.019938
2000-01-07 -0.559474  0.106248 -0.157005 -0.301423
2000-01-08 -0.362920  0.158875 -0.338801 -0.407528
2000-01-09 -0.478920  0.042699 -0.269751 -0.267903
2000-01-10 -0.378724 -0.095881 -0.019401 -0.327291


                   A         B         C         D
2000-01-01  1.405652  0.439059 -0.818007  0.096281
2000-01-02  0.792898  0.524493 -0.507557 -0.539432
2000-01-03 -0.965274  0.189189  0.632896 -0.617476
2000-01-04 -1.282158 -0.464220 -0.087011  0.535104
2000-01-05 -0.644676  0.567123  0.116558  0.593682
2000-01-06 -0.947493  0.001799 -0.998898 -0.153175
2000-01-07 -1.183308 -0.203735  0.170340 -1.378510
2000-01-08  0.281091  0.283672 -1.017647 -1.226320
2000-01-09 -0.844304 -0.496620 -0.150741  0.157358
2000-01-10  0.067271 -1.060953  1.438944 -0.522078
```

Aggregations

In [ ]:

```
df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])

print (df)
r = df.rolling(window=3,min_periods=1)
print (r)
print("\n")
print (r.aggregate(np.sum)) #Apply Aggregation on a Whole Dataframe
print("\n")
print(r['A'].aggregate(np.sum)) #Apply Aggregation on a Single Column of a Dataframe
print("\n")
print(r[['A','B']].aggregate(np.sum)) #Apply Aggregation on Multiple Columns of a DataFra
me
print("\n")
print(r['A'].aggregate([np.sum,np.mean])) #Apply Multiple Functions on a Single Column of
a DataFrame
print("\n")
print(r[['A','B']].aggregate([np.sum,np.mean])) #Apply Multiple Functions on Multiple Col
umns of a DataFrame
print("\n","Apply Different Functions to Different Columns of a Dataframe")
print (r.aggregate({'A' : np.sum,'B' : np.mean}))
```

```
                   A         B         C         D
2000-01-01 -0.635177  1.524684  0.034425 -0.901272
2000-01-02  1.063565  0.984277  0.337346  0.650422
2000-01-03 -0.036745 -0.226086  0.628010 -0.261269
2000-01-04 -0.991549  1.440361 -1.213301 -0.497648
2000-01-05  0.197667 -0.744149  1.624848 -0.312195
2000-01-06  0.689463 -0.583654  0.893779 -0.505081
2000-01-07 -1.093670  1.600719 -0.866959 -0.957218
2000-01-08 -2.059971  0.866450  1.086146  0.338824
2000-01-09 -1.567148  0.145687  1.805380 -1.097779
2000-01-10 -1.809699 -1.394781 -0.871341 -0.588241
Rolling [window=3,min_periods=1,center=False,axis=0]


                   A         B         C         D
2000-01-01 -0.635177  1.524684  0.034425 -0.901272
2000-01-02  0.428388  2.508960  0.371771 -0.250850
2000-01-03  0.391643  2.282874  0.999781 -0.512119
2000-01-04  0.035272  2.198551 -0.247945 -0.108496
2000-01-05 -0.830626  0.470126  1.039557 -1.071113
2000-01-06 -0.104419  0.112559  1.305326 -1.314924
2000-01-07 -0.206541  0.272916  1.651668 -1.774494
2000-01-08 -2.464179  1.883515  1.112966 -1.123474
2000-01-09 -4.720789  2.612855  2.024567 -1.716173
2000-01-10 -5.436818 -0.382644  2.020185 -1.347197


2000-01-01   -0.635177
2000-01-02    0.428388
2000-01-03    0.391643
2000-01-04    0.035272
2000-01-05   -0.830626
2000-01-06   -0.104419
2000-01-07   -0.206541
2000-01-08   -2.464179
2000-01-09   -4.720789
2000-01-10   -5.436818
Freq: D, Name: A, dtype: float64


                   A         B
2000-01-01 -0.635177  1.524684
2000-01-02  0.428388  2.508960
2000-01-03  0.391643  2.282874
2000-01-04  0.035272  2.198551
2000-01-05 -0.830626  0.470126
2000-01-06 -0.104419  0.112559
2000-01-07 -0.206541  0.272916
2000-01-08 -2.464179  1.883515
```

```
2000-01-08 -2.464179  1.883515
2000-01-09 -4.720789  2.612855
2000-01-10 -5.436818 -0.382644


                 sum      mean
2000-01-01 -0.635177 -0.635177
2000-01-02  0.428388  0.214194
2000-01-03  0.391643  0.130548
2000-01-04  0.035272  0.011757
2000-01-05 -0.830626 -0.276875
2000-01-06 -0.104419 -0.034806
2000-01-07 -0.206541 -0.068847
2000-01-08 -2.464179 -0.821393
2000-01-09 -4.720789 -1.573596
2000-01-10 -5.436818 -1.812273


                   A                   B
                 sum      mean      sum      mean
2000-01-01 -0.635177 -0.635177  1.524684  1.524684
2000-01-02  0.428388  0.214194  2.508960  1.254480
2000-01-03  0.391643  0.130548  2.282874  0.760958
2000-01-04  0.035272  0.011757  2.198551  0.732850
2000-01-05 -0.830626 -0.276875  0.470126  0.156709
2000-01-06 -0.104419 -0.034806  0.112559  0.037520
2000-01-07 -0.206541 -0.068847  0.272916  0.090972
2000-01-08 -2.464179 -0.821393  1.883515  0.627838
2000-01-09 -4.720789 -1.573596  2.612855  0.870952
2000-01-10 -5.436818 -1.812273 -0.382644 -0.127548

 Apply Different Functions to Different Columns of a Dataframe
                   A         B
2000-01-01 -0.635177  1.524684
2000-01-02  0.428388  1.254480
2000-01-03  0.391643  0.760958
2000-01-04  0.035272  0.732850
2000-01-05 -0.830626  0.156709
2000-01-06 -0.104419  0.037520
2000-01-07 -0.206541  0.090972
2000-01-08 -2.464179  0.627838
2000-01-09 -4.720789  0.870952
2000-01-10 -5.436818 -0.127548
```

Missing Data

In [ ]:

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print(df)
```

```
        one       two     three
a -1.195818  0.380775  1.081202
b       NaN       NaN       NaN
c  0.085767  0.326588  2.330922
d       NaN       NaN       NaN
e  0.549948 -0.220830 -1.559583
f  1.408340  0.266646 -0.094768
g       NaN       NaN       NaN
h  1.859332  1.666616  0.208227
```

Check for Missing Values

Pandas provides the isnull() and notnull() functions, which are also methods on Series and DataFrame objects

In [ ]:

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print (df['one'].isnull())
print("\n")
print (df['one'].notnull())
```

```
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool


a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
Name: one, dtype: bool
```

Calculations with Missing Data

- **When summing data, NA will be treated as Zero**
- **If the data are all NA, then the result will be NA**

In [ ]:

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print (df['one'].sum())
```

```
-2.204395666191723
```

In [ ]:

```
df = pd.DataFrame(index=[0,1,2,3,4,5],columns=['one','two'])
print (df['one'].sum())
```

```
0
```

Cleaning / Filling Missing Data

Replace NaN with a Scalar Value

In [ ]:

```
df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c', 'e'],columns=['one',
'two', 'three'])

df = df.reindex(['a', 'b', 'c'])

print (df)
print ("NaN replaced with '0':")
```

```
print (df.fillna(0))
```

```
        one       two     three
a  0.342564 -0.775108 -0.959250
b       NaN       NaN       NaN
c  0.060223 -1.881292 -0.836699
NaN replaced with '0':
        one       two     three
a  0.342564 -0.775108 -0.959250
b  0.000000  0.000000  0.000000
c  0.060223 -1.881292 -0.836699
```

Fill NA Forward and Backward

Sr.No	Method & Action
1	**pad/fill** Fill methods Forward
2	**bfill/backfill** Fill methods Backward

In [ ]:

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print (df.fillna(method='pad'))
print (df.fillna(method='backfill'))
```

```
        one       two     three
a -1.334556 -0.296370 -1.635900
b -1.334556 -0.296370 -1.635900
c  0.888231 -1.658487 -0.804168
d  0.888231 -1.658487 -0.804168
e -0.669128  0.863487  1.106005
f  0.440386 -0.742258  1.656160
g  0.440386 -0.742258  1.656160
h -0.439241 -1.212085 -0.125723
        one       two     three
a -1.334556 -0.296370 -1.635900
b  0.888231 -1.658487 -0.804168
c  0.888231 -1.658487 -0.804168
d -0.669128  0.863487  1.106005
e -0.669128  0.863487  1.106005
f  0.440386 -0.742258  1.656160
g -0.439241 -1.212085 -0.125723
h -0.439241 -1.212085 -0.125723
```

Drop Missing Values

In [ ]:

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
print(df)

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
```

```
print (df.dropna())
print (df.dropna(axis=1))
```

```
        one       two     three
a  0.135488 -0.258351  0.978610
c -0.897853 -0.183906  1.458668
e -0.298103  0.795800 -0.119068
f  0.481650  0.560387  0.543059
h  1.114430 -0.485046  0.082327
        one       two     three
a  0.135488 -0.258351  0.978610
c -0.897853 -0.183906  1.458668
e -0.298103  0.795800 -0.119068
f  0.481650  0.560387  0.543059
h  1.114430 -0.485046  0.082327
Empty DataFrame
Columns: []
Index: [a, b, c, d, e, f, g, h]
```

Replace Missing (or) Generic Values

In [ ]:

```
df = pd.DataFrame({'one':[10,20,30,40,50,2000], 'two':[1000,0,30,40,50,60]})

print (df.replace({1000:10,2000:60}))
```

```
   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
5   60   60
```

In [ ]:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000], 'two':[1000,0,30,40,50,60]})
print (df.replace({1000:10,2000:60}))
```

```
   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
5   60   60
```

GroupBy

Splitting the Object

Applying a function

Combining the results

Aggregation – computing a summary statistic

Transformation – perform some group-specific operation

Filtration – discarding the data with some condition

In [ ]:

```
ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',
```

```python
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

print (df)
print("\n")
print (df.groupby('Team')) #Split Data into Groups
print("\n")
print(df.groupby('Team').groups) #view groups
print("\n")
print(df.groupby(['Team','Year']).groups) #Group by with multiple columns
print("\n")

grouped = df.groupby('Year') #iterating through groups

for name,group in grouped:
   print (name)
   print (group)

grouped = df.groupby('Year') #select a group
print (grouped.get_group(2014))
```

```
      Team  Rank  Year  Points
0   Riders     1  2014     876
1   Riders     2  2015     789
2   Devils     2  2014     863
3   Devils     3  2015     673
4    Kings     3  2014     741
5    kings     4  2015     812
6    Kings     1  2016     756
7    Kings     1  2017     788
8   Riders     2  2016     694
9   Royals     4  2014     701
10  Royals     1  2015     804
11  Riders     2  2017     690


<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f70e5132410>


{'Devils': [2, 3], 'Kings': [4, 6, 7], 'Riders': [0, 1, 8, 11], 'Royals': [9, 10], 'kings
': [5]}


{('Devils', 2014): [2], ('Devils', 2015): [3], ('Kings', 2014): [4], ('Kings', 2016): [6]
, ('Kings', 2017): [7], ('Riders', 2014): [0], ('Riders', 2015): [1], ('Riders', 2016): [
8], ('Riders', 2017): [11], ('Royals', 2014): [9], ('Royals', 2015): [10], ('kings', 2015
): [5]}


2014
     Team  Rank  Year  Points
0  Riders     1  2014     876
2  Devils     2  2014     863
4   Kings     3  2014     741
9  Royals     4  2014     701
2015
      Team  Rank  Year  Points
1   Riders     2  2015     789
3   Devils     3  2015     673
5    kings     4  2015     812
10  Royals     1  2015     804
2016
     Team  Rank  Year  Points
6   Kings     1  2016     756
8  Riders     2  2016     694
2017
      Team  Rank  Year  Points
7    Kings     1  2017     788
11  Riders     2  2017     690
```

```
     Team  Rank  Year  Points
0  Riders     1  2014     876
2  Devils     2  2014     863
4   Kings     3  2014     741
9  Royals     4  2014     701
```

Merging/Joining

pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=True)

left – A DataFrame object.

right – Another DataFrame object.

on – Columns (names) to join on. Must be found in both the left and right DataFrame objects.

left_on – Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame.

right_on – Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame.

left_index – If True, use the index (row labels) from the left DataFrame as its join key(s). In case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame.

right_index – Same usage as left_index for the right DataFrame.

how – One of 'left', 'right', 'outer', 'inner'. Defaults to inner. Each method has been described below.

sort – Sort the result DataFrame by the join keys in lexicographical order. Defaults to True, setting to False will improve the performance substantially in many cases.

In [9]:

```
left = pd.DataFrame({
   'id':[1,2,3,4,5],
   'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
   'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right = pd.DataFrame({
 'id':[1,2,3,4,5],
   'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
   'subject_id':['sub2','sub4','sub3','sub6','sub5']})
print (pd.merge(left,right,on=['id','subject_id']))
```

```
   id  Name_x subject_id Name_y
0   4   Alice       sub6  Bryce
1   5  Ayoung       sub5  Betty
```

Merge Method	SQL Equivalent	Description
left	LEFT OUTER JOIN	Use keys from left object
right	RIGHT OUTER JOIN	Use keys from right object
outer	FULL OUTER JOIN	Use union of keys
inner	INNER JOIN	Use intersection of keys