Groupby

Any **groupby** operation involves one of the following operations on the original object. They are −

- **Splitting** the Object
- **Applying** a function
- **Combining** the results

In many situations, we split the data into sets and we apply some functionality on each subset. In the apply functionality, we can perform the following operations −

- **Aggregation** − computing a summary statistic
- **Transformation** − perform some group-specific operation
- **Filtration** − discarding the data with some condition

Let us now create a DataFrame object and perform all the operations on it −

Live Demo

```
#import the pandas library
import pandas as pd

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

Its **output** is as follows −

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

Split Data into Groups

Pandas object can be split into any of their objects. There are multiple ways to split an object like −

- obj.groupby('key')
- obj.groupby(['key1','key2'])
- obj.groupby(key,axis=1)

Let us now see how the grouping objects can be applied to the DataFrame object

Example

Live Demo

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 
'Kings',
   '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.groupby('Team')
```

Its **output** is as follows −

```
<pandas.core.groupby.DataFrameGroupBy object at 0x7fa46a977e50>
```

View Groups

Live Demo

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 
'Kings',
   '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.groupby('Team').groups
```

Its **output** is as follows −

```
{'Kings': Int64Index([4, 6, 7],      dtype='int64'),
'Devils': Int64Index([2, 3],         dtype='int64'),
'Riders': Int64Index([0, 1, 8, 11],  dtype='int64'),
'Royals': Int64Index([9, 10],        dtype='int64'),
'kings' : Int64Index([5],            dtype='int64')}
```

Example

Group by with multiple columns −

Live Demo

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   '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.groupby(['Team','Year']).groups
```

Its **output** is as follows −

```
{('Kings', 2014): Int64Index([4], dtype='int64'),
 ('Royals', 2014): Int64Index([9], dtype='int64'),
 ('Riders', 2014): Int64Index([0], dtype='int64'),
 ('Riders', 2015): Int64Index([1], dtype='int64'),
 ('Kings', 2016): Int64Index([6], dtype='int64'),
 ('Riders', 2016): Int64Index([8], dtype='int64'),
 ('Riders', 2017): Int64Index([11], dtype='int64'),
 ('Devils', 2014): Int64Index([2], dtype='int64'),
 ('Devils', 2015): Int64Index([3], dtype='int64'),
 ('kings', 2015): Int64Index([5], dtype='int64'),
 ('Royals', 2015): Int64Index([10], dtype='int64'),
 ('Kings', 2017): Int64Index([7], dtype='int64')}
```

Iterating through Groups

With the **groupby** object in hand, we can iterate through the object similar to itertools.obj.

Live Demo

```
# import the pandas library
import pandas as pd

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

grouped = df.groupby('Year')

for name,group in grouped:
   print name
   print group
```

Its **output** is as follows −

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

2015
   Points   Rank     Team   Year
1     789      2   Riders   2015
3     673      3   Devils   2015
5     812      4    kings   2015
10    804      1   Royals   2015

2016
   Points   Rank     Team   Year
6     756      1    Kings   2016
8     694      2   Riders   2016

2017
   Points   Rank    Team   Year
7     788      1   Kings   2017
11    690      2  Riders   2017
```

By default, the **groupby** object has the same label name as the group name.

Select a Group

Using the **get_group()** method, we can select a single group.

Live Demo

```
# import the pandas library
import pandas as pd

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

grouped = df.groupby('Year')
print grouped.get_group(2014)
```

Its **output** is as follows −

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

Aggregations

An aggregated function returns a single aggregated value for each group. Once the **group by** object is created, several aggregation operations can be performed on the grouped data.

An obvious one is aggregation via the aggregate or equivalent **agg** method −

Live Demo

```
# import the pandas library
import pandas as pd
import numpy as np

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

```
grouped = df.groupby('Year')
print grouped['Points'].agg(np.mean)
```

Its **output** is as follows −

```
Year
2014   795.25
2015   769.50
2016   725.00
2017   739.00
Name: Points, dtype: float64
```

Another way to see the size of each group is by applying the size() function −

Live Demo

```
import pandas as pd
import numpy as np

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

Attribute Access in Python Pandas
grouped = df.groupby('Team')
print grouped.agg(np.size)
```

Its **output** is as follows −

```
         Points   Rank   Year
Team
Devils        2      2      2
Kings         3      3      3
Riders        4      4      4
Royals        2      2      2
kings         1      1      1
```

Applying Multiple Aggregation Functions at Once

With grouped Series, you can also pass a **list** or **dict of functions** to do aggregation with, and generate DataFrame as output −

Live Demo

```
# import the pandas library
import pandas as pd
import numpy as np
```

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

grouped = df.groupby('Team')
print grouped['Points'].agg([np.sum, np.mean, np.std])
```

Its **output** is as follows −

```
Team      sum      mean          std
Devils   1536   768.000000   134.350288
Kings    2285   761.666667    24.006943
Riders   3049   762.250000    88.567771
Royals   1505   752.500000    72.831998
kings     812   812.000000          NaN
```

Transformations

Transformation on a group or a column returns an object that is indexed the same size of that is being grouped. Thus, the transform should return a result that is the same size as that of a group chunk.

Live Demo

```
# import the pandas library
import pandas as pd
import numpy as np

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

grouped = df.groupby('Team')
score = lambda x: (x - x.mean()) / x.std()*10
print grouped.transform(score)
```

Its **output** is as follows −

```
       Points        Rank        Year
0   12.843272  -15.000000  -11.618950
1    3.020286    5.000000   -3.872983
2    7.071068   -7.071068   -7.071068
3   -7.071068    7.071068    7.071068
4   -8.608621   11.547005  -10.910895
5         NaN         NaN         NaN
6   -2.360428   -5.773503    2.182179
7   10.969049   -5.773503    8.728716
8   -7.705963    5.000000    3.872983
9   -7.071068    7.071068   -7.071068
10   7.071068   -7.071068    7.071068
11  -8.157595    5.000000   11.618950
```

Filtration

Filtration filters the data on a defined criteria and returns the subset of data. The **filter()** function is used to filter the data.

Live Demo

```
import pandas as pd
import numpy as np

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   '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.groupby('Team').filter(lambda x: len(x) >= 3)
```

Its **output** is as follows −

```
    Points  Rank     Team   Year
0      876     1   Riders   2014
1      789     2   Riders   2015
4      741     3   Kings    2014
6      756     1   Kings    2016
7      788     1   Kings    2017
8      694     2   Riders   2016
11     690     2   Riders   2017
```

In the above filter condition, we are asking to return the teams which have participated three or more times in IPL.

Merging/Joining

Pandas has full-featured, high performance in-memory join operations idiomatically very similar to relational databases like SQL.

Pandas provides a single function, **merge**, as the entry point for all standard database join operations between DataFrame objects −

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

Here, we have used the following parameters −

- **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.

Let us now create two different DataFrames and perform the merging operations on it.

Live Demo

```
# import the pandas library
import pandas as pd
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 left
print right
```

Its **output** is as follows −

```
     Name  id   subject_id
0    Alex   1         sub1
1     Amy   2         sub2
2   Allen   3         sub4
3   Alice   4         sub6
4  Ayoung   5         sub5

     Name  id   subject_id
0   Billy   1         sub2
1   Brian   2         sub4
2   Bran    3         sub3
3   Bryce   4         sub6
4   Betty   5         sub5
```

Merge Two DataFrames on a Key

Live Demo

```
import pandas as pd
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')
```

Its **output** is as follows −

```
   Name_x   id  subject_id_x   Name_y   subject_id_y
0  Alex      1          sub1    Billy           sub2
1  Amy       2          sub2    Brian           sub4
2  Allen     3          sub4     Bran           sub3
3  Alice     4          sub6    Bryce           sub6
4  Ayoung    5          sub5    Betty           sub5
```

Merge Two DataFrames on Multiple Keys

Live Demo

```
import pandas as pd
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'])
```

Its **output** is as follows −

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

Merge Using 'how' Argument

The **how** argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination does not appear in either the left or the right tables, the values in the joined table will be NA.

Here is a summary of the **how** options and their SQL equivalent names −

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

Left Join

Live Demo

```
import pandas as pd
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='subject_id', how='left')
```

Its **output** is as follows −

```
    Name_x   id_x   subject_id   Name_y   id_y
0     Alex      1         sub1      NaN    NaN
```

```
1       Amy       2          sub2    Billy    1.0
2     Allen       3          sub4    Brian    2.0
3     Alice       4          sub6    Bryce    4.0
4    Ayoung       5          sub5    Betty    5.0
```

Right Join

Live Demo

```python
import pandas as pd
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='subject_id', how='right')
```

Its **output** is as follows −

```
    Name_x  id_x   subject_id   Name_y   id_y
0      Amy   2.0         sub2    Billy      1
1    Allen   3.0         sub4    Brian      2
2    Alice   4.0         sub6    Bryce      4
3   Ayoung   5.0         sub5    Betty      5
4      NaN   NaN         sub3     Bran      3
```

Outer Join

Live Demo

```python
import pandas as pd
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, how='outer', on='subject_id')
```

Its **output** is as follows −

```
    Name_x  id_x   subject_id   Name_y   id_y
0     Alex   1.0         sub1      NaN    NaN
1      Amy   2.0         sub2    Billy    1.0
2    Allen   3.0         sub4    Brian    2.0
3    Alice   4.0         sub6    Bryce    4.0
4   Ayoung   5.0         sub5    Betty    5.0
```

```
5       NaN    NaN           sub3      Bran     3.0
```

Inner Join

Joining will be performed on index. Join operation honors the object on which it is called. So, **a.join(b)** is not equal to **b.join(a)**.

Live Demo

```
import pandas as pd
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='subject_id', how='inner')
```

Its **output** is as follows −

```
    Name_x   id_x   subject_id   Name_y   id_y
0      Amy      2         sub2    Billy      1
1    Allen      3         sub4    Brian      2
2    Alice      4         sub6    Bryce      4
3   Ayoung      5         sub5    Betty      5
```

Concatenation

Pandas provides various facilities for easily combining together **Series, DataFrame**, and **Panel** objects.

```
 pd.concat(objs,axis=0,join='outer',join_axes=None,
ignore_index=False)
```

- **objs** − This is a sequence or mapping of Series, DataFrame, or Panel objects.
- **axis** − {0, 1, ...}, default 0. This is the axis to concatenate along.
- **join** − {'inner', 'outer'}, default 'outer'. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- **ignore_index** − boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1.
- **join_axes** − This is the list of Index objects. Specific indexes to use for the other (n-1) axes instead of performing inner/outer set logic.

Concatenating Objects

The **concat** function does all of the heavy lifting of performing concatenation operations along an axis. Let us create different objects and do concatenation.

Live Demo

```
import pandas as pd

one = pd.DataFrame({
   'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
   'subject_id':['sub1','sub2','sub4','sub6','sub5'],
   'Marks_scored':[98,90,87,69,78]},
   index=[1,2,3,4,5])

two = pd.DataFrame({
   'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
   'subject_id':['sub2','sub4','sub3','sub6','sub5'],
   'Marks_scored':[89,80,79,97,88]},
   index=[1,2,3,4,5])
print pd.concat([one,two])
```

Its **output** is as follows −

```
    Marks_scored     Name   subject_id
1             98     Alex         sub1
2             90      Amy         sub2
3             87    Allen         sub4
4             69    Alice         sub6
5             78   Ayoung         sub5
1             89    Billy         sub2
2             80    Brian         sub4
3             79     Bran         sub3
4             97    Bryce         sub6
5             88    Betty         sub5
```

Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this by using the **keys** argument −

Live Demo

```
import pandas as pd

one = pd.DataFrame({
   'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
   'subject_id':['sub1','sub2','sub4','sub6','sub5'],
   'Marks_scored':[98,90,87,69,78]},
   index=[1,2,3,4,5])

two = pd.DataFrame({
   'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
   'subject_id':['sub2','sub4','sub3','sub6','sub5'],
   'Marks_scored':[89,80,79,97,88]},
   index=[1,2,3,4,5])
print pd.concat([one,two],keys=['x','y'])
```

Its **output** is as follows −

```
x  1  98     Alex     sub1
   2  90     Amy      sub2
   3  87     Allen    sub4
   4  69     Alice    sub6
   5  78     Ayoung   sub5
y  1  89     Billy    sub2
   2  80     Brian    sub4
   3  79     Bran     sub3
   4  97     Bryce    sub6
   5  88     Betty    sub5
```

The index of the resultant is duplicated; each index is repeated.

If the resultant object has to follow its own indexing, set **ignore_index** to **True**.

Live Demo

```
import pandas as pd

one = pd.DataFrame({
   'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
   'subject_id':['sub1','sub2','sub4','sub6','sub5'],
   'Marks_scored':[98,90,87,69,78]},
   index=[1,2,3,4,5])

two = pd.DataFrame({
   'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
   'subject_id':['sub2','sub4','sub3','sub6','sub5'],
   'Marks_scored':[89,80,79,97,88]},
   index=[1,2,3,4,5])
print pd.concat([one,two],keys=['x','y'],ignore_index=True)
```

Its **output** is as follows −

```
    Marks_scored     Name    subject_id
0             98     Alex          sub1
1             90      Amy          sub2
2             87    Allen          sub4
3             69    Alice          sub6
4             78   Ayoung          sub5
5             89    Billy          sub2
6             80    Brian          sub4
7             79     Bran          sub3
8             97    Bryce          sub6
9             88    Betty          sub5
```

Observe, the index changes completely and the Keys are also overridden.

If two objects need to be added along **axis=1**, then the new columns will be appended.

Live Demo

```
import pandas as pd

one = pd.DataFrame({
   'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
   'subject_id':['sub1','sub2','sub4','sub6','sub5'],
   'Marks_scored':[98,90,87,69,78]},
   index=[1,2,3,4,5])

two = pd.DataFrame({
   'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
   'subject_id':['sub2','sub4','sub3','sub6','sub5'],
   'Marks_scored':[89,80,79,97,88]},
   index=[1,2,3,4,5])
print pd.concat([one,two],axis=1)
```

Its **output** is as follows −

```
    Marks_scored     Name   subject_id   Marks_scored    Name
subject_id
1           98       Alex       sub1           89         Billy
sub2
2           90        Amy       sub2           80         Brian
sub4
3           87      Allen       sub4           79          Bran
sub3
4           69      Alice       sub6           97         Bryce
sub6
5           78     Ayoung       sub5           88         Betty
sub5
```

Concatenating Using append

A useful shortcut to concat are the append instance methods on Series and DataFrame. These methods actually predated concat. They concatenate along **axis=0**, namely the index −

Live Demo

```
import pandas as pd

one = pd.DataFrame({
   'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
   'subject_id':['sub1','sub2','sub4','sub6','sub5'],
   'Marks_scored':[98,90,87,69,78]},
   index=[1,2,3,4,5])

two = pd.DataFrame({
   'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
   'subject_id':['sub2','sub4','sub3','sub6','sub5'],
   'Marks_scored':[89,80,79,97,88]},
   index=[1,2,3,4,5])
```

```
print one.append(two)
```

Its **output** is as follows −

```
    Marks_scored     Name   subject_id
1             98     Alex       sub1
2             90      Amy       sub2
3             87    Allen       sub4
4             69    Alice       sub6
5             78   Ayoung       sub5
1             89    Billy       sub2
2             80    Brian       sub4
3             79     Bran       sub3
4             97    Bryce       sub6
5             88    Betty       sub5
```

The **append** function can take multiple objects as well −

Live Demo

```
import pandas as pd

one = pd.DataFrame({
   'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
   'subject_id':['sub1','sub2','sub4','sub6','sub5'],
   'Marks_scored':[98,90,87,69,78]},
   index=[1,2,3,4,5])

two = pd.DataFrame({
   'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
   'subject_id':['sub2','sub4','sub3','sub6','sub5'],
   'Marks_scored':[89,80,79,97,88]},
   index=[1,2,3,4,5])
print one.append([two,one,two])
```

Its **output** is as follows −

```
    Marks_scored    Name    subject_id
1             98    Alex          sub1
2             90     Amy          sub2
3             87   Allen          sub4
4             69   Alice          sub6
5             78  Ayoung          sub5
1             89   Billy          sub2
2             80   Brian          sub4
3             79    Bran          sub3
4             97   Bryce          sub6
5             88   Betty          sub5
1             98    Alex          sub1
2             90     Amy          sub2
3             87   Allen          sub4
4             69   Alice          sub6
5             78  Ayoung          sub5
```

```
1            89    Billy          sub2
2            80    Brian          sub4
3            79     Bran          sub3
4            97    Bryce          sub6
5            88    Betty          sub5
```

Time Series

Pandas provide a robust tool for working time with Time series data, especially in the financial sector. While working with time series data, we frequently come across the following −

- Generating sequence of time
- Convert the time series to different frequencies

Pandas provides a relatively compact and self-contained set of tools for performing the above tasks.

Get Current Time

datetime.now() gives you the current date and time.

Live Demo

```
import pandas as pd

print pd.datetime.now()
```

Its **output** is as follows −

```
2017-05-11 06:10:13.393147
```

Create a TimeStamp

Time-stamped data is the most basic type of timeseries data that associates values with points in time. For pandas objects, it means using the points in time. Let's take an example −

Live Demo

```
import pandas as pd

print pd.Timestamp('2017-03-01')
```

Its **output** is as follows −

```
2017-03-01 00:00:00
```

It is also possible to convert integer or float epoch times. The default unit for these is nanoseconds (since these are how Timestamps are stored). However, often epochs are stored in another unit which can be specified. Let's take another example

Live Demo

```
import pandas as pd

print pd.Timestamp(1587687255,unit='s')
```

Its **output** is as follows −

```
2020-04-24 00:14:15
```

Create a Range of Time

Live Demo

```
import pandas as pd

print pd.date_range("11:00", "13:30", freq="30min").time
```

Its **output** is as follows −

```
[datetime.time(11, 0) datetime.time(11, 30) datetime.time(12, 0)
datetime.time(12, 30) datetime.time(13, 0) datetime.time(13, 30)]
```

Change the Frequency of Time

Live Demo

```
import pandas as pd

print pd.date_range("11:00", "13:30", freq="H").time
```

Its **output** is as follows −

```
[datetime.time(11, 0) datetime.time(12, 0) datetime.time(13, 0)]
```

Converting to Timestamps

To convert a Series or list-like object of date-like objects, for example strings, epochs, or a mixture, you can use the **to_datetime** function. When passed, this returns a Series (with the same index), while a **list-like** is converted to a **DatetimeIndex**. Take a look at the following example −

Live Demo

```
import pandas as pd

print pd.to_datetime(pd.Series(['Jul 31, 2009','2010-01-10',
None]))
```

Its **output** is as follows −

```
0   2009-07-31
1   2010-01-10
2          NaT
dtype: datetime64[ns]
```

NaT means **Not a Time** (equivalent to NaN)

Let's take another example.

Live Demo

```
import pandas as pd

print pd.to_datetime(['2005/11/23', '2010.12.31', None])
```

Its **output** is as follows −

```
DatetimeIndex(['2005-11-23', '2010-12-31', 'NaT'],
dtype='datetime64[ns]', freq=None
```

TimeDelta

Timedeltas are differences in times, expressed in difference units, for example, days, hours, minutes, seconds. They can be both positive and negative.

We can create Timedelta objects using various arguments as shown below −

String

By passing a string literal, we can create a timedelta object.

Live Demo

```
import pandas as pd

print pd.Timedelta('2 days 2 hours 15 minutes 30 seconds')
```

Its **output** is as follows −

```
2 days 02:15:30
```

Integer

By passing an integer value with the unit, an argument creates a Timedelta object.

Live Demo

```
import pandas as pd

print pd.Timedelta(6,unit='h')
```

Its **output** is as follows −

```
0 days 06:00:00
```

Data Offsets

Data offsets such as - weeks, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds can also be used in construction.

Live Demo

```
import pandas as pd

print pd.Timedelta(days=2)
```

Its **output** is as follows −

```
2 days 00:00:00
```

to_timedelta()

Using the top-level **pd.to_timedelta**, you can convert a scalar, array, list, or series from a recognized timedelta format/ value into a Timedelta type. It will construct Series if the input is a Series, a scalar if the input is scalar-like, otherwise will output a **TimedeltaIndex**.

Live Demo

```
import pandas as pd

print pd.Timedelta(days=2)
```

Its **output** is as follows −

```
2 days 00:00:00
```

Operations

You can operate on Series/ DataFrames and construct **timedelta64[ns]** Series through subtraction operations on **datetime64[ns]** Series, or Timestamps.

Let us now create a DataFrame with Timedelta and datetime objects and perform some arithmetic operations on it −

Live Demo

```
import pandas as pd

s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))
td = pd.Series([ pd.Timedelta(days=i) for i in range(3) ])
df = pd.DataFrame(dict(A = s, B = td))
```

```
print df
```

Its **output** is as follows −

```
              A       B
0   2012-01-01 0 days
1   2012-01-02 1 days
2   2012-01-03 2 days
```

Addition Operations

Live Demo

```
import pandas as pd

s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))
td = pd.Series([ pd.Timedelta(days=i) for i in range(3) ])
df = pd.DataFrame(dict(A = s, B = td))
df['C']=df['A']+df['B']

print df
```

Its **output** is as follows −

```
           A      B          C
0 2012-01-01 0 days 2012-01-01
1 2012-01-02 1 days 2012-01-03
2 2012-01-03 2 days 2012-01-05
```

Subtraction Operation

Live Demo

```
import pandas as pd

s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))
td = pd.Series([ pd.Timedelta(days=i) for i in range(3) ])
df = pd.DataFrame(dict(A = s, B = td))
df['C']=df['A']+df['B']
df['D']=df['C']+df['B']

print df
```

Its **output** is as follows −

```
           A      B          C          D
0 2012-01-01 0 days 2012-01-01 2012-01-01
1 2012-01-02 1 days 2012-01-03 2012-01-04
2 2012-01-03 2 days 2012-01-05 2012-01-07
```