Statistical Functions

Statistical methods help in the understanding and analyzing the behavior of data. We will now learn a few statistical functions, which we can apply on Pandas objects.

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.

Live Demo

```
import pandas as pd
import numpy as np
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()
```

Its **output** is as follows −

```
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  -15.151902    0.174730
2   -0.746374   -1.449088
3   -3.582229   -3.165836
4    15.601150  -1.860434
```

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.

Cov Series

Live Demo

```
import pandas as pd
```

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

Its **output** is as follows −

```
-0.12978405324
```

Covariance method when applied on a DataFrame, computes **cov** between all the columns.

Live Demo

```
import pandas as pd
import numpy as np
frame = pd.DataFrame(np.random.randn(10, 5), columns=['a', 'b',
'c', 'd', 'e'])
print frame['a'].cov(frame['b'])
print frame.cov()
```

Its **output** is as follows −

```
-0.58312921152741437

          a         b         c         d         e
a  1.780628 -0.583129 -0.185575  0.003679 -0.136558
b -0.583129  1.297011  0.136530 -0.523719  0.251064
c -0.185575  0.136530  0.915227 -0.053881 -0.058926
d  0.003679 -0.523719 -0.053881  1.521426 -0.487694
e -0.136558  0.251064 -0.058926 -0.487694  0.960761
```

Note − Observe the **cov** between **a** and **b** column in the first statement and the same is the value returned by cov on DataFrame.

Correlation

Correlation shows the linear relationship between any two array of values (series). There are multiple methods to compute the correlation like pearson(default), spearman and kendall.

Live Demo

```
import pandas as pd
import numpy as np
frame = pd.DataFrame(np.random.randn(10, 5), columns=['a', 'b',
'c', 'd', 'e'])

print frame['a'].corr(frame['b'])
print frame.corr()
```

Its **output** is as follows −

```
-0.383712785514
```

```
          a         b         c         d         e
a  1.000000 -0.383713 -0.145368  0.002235 -0.104405
b -0.383713  1.000000  0.125311 -0.372821  0.224908
c -0.145368  0.125311  1.000000 -0.045661 -0.062840
d  0.002235 -0.372821 -0.045661  1.000000 -0.403380
e -0.104405  0.224908 -0.062840 -0.403380  1.000000
```

If any non-numeric column is present in the DataFrame, it is excluded automatically.

Data Ranking

Data Ranking produces ranking for each element in the array of elements. In case of ties, assigns the mean rank.

Live Demo

```
import pandas as pd
import numpy as np

s = pd.Series(np.random.np.random.randn(5), index=list('abcde'))
s['d'] = s['b'] # so there's a tie
print s.rank()
```

Its **output** is as follows −

```
a  1.0
b  3.5
c  2.0
d  3.5
e  5.0
dtype: float64
```

Rank optionally takes a parameter ascending which by default is true; when false, data is reverse-ranked, with larger values assigned a smaller rank.

Rank supports different tie-breaking methods, specified with the method parameter −

- **average** − average rank of tied group
- **min** − lowest rank in the group
- **max** − highest rank in the group
- **first** − ranks assigned in the order they appear in the array

Window Functions

For working on numerical data, Pandas provide few variants like rolling, expanding and exponentially moving weights for window statistics. Among these are **sum, mean, median, variance, covariance, correlation,** etc.

We will now learn how each of these can be applied on DataFrame objects.

.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.

Live Demo

```
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()
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01       NaN         NaN         NaN         NaN
2000-01-02       NaN         NaN         NaN         NaN
2000-01-03  0.434553   -0.667940   -1.051718   -0.826452
2000-01-04  0.628267   -0.047040   -0.287467   -0.161110
2000-01-05  0.398233    0.003517    0.099126   -0.405565
2000-01-06  0.641798    0.656184   -0.322728    0.428015
2000-01-07  0.188403    0.010913   -0.708645    0.160932
2000-01-08  0.188043   -0.253039   -0.818125   -0.108485
2000-01-09  0.682819   -0.606846   -0.178411   -0.404127
2000-01-10  0.688583    0.127786    0.513832   -1.067156
```

Note − Since the window size is 3, for first two elements there are nulls and from third the value will be the average of the **n**, **n-1** and **n-2** elements. Thus we can also apply various functions as mentioned above.

.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.

Live Demo

```
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.expanding(min_periods=3).mean()
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01       NaN         NaN         NaN         NaN
2000-01-02       NaN         NaN         NaN         NaN
2000-01-03  0.434553   -0.667940   -1.051718   -0.826452
2000-01-04  0.743328   -0.198015   -0.852462   -0.262547
2000-01-05  0.614776   -0.205649   -0.583641   -0.303254
2000-01-06  0.538175   -0.005878   -0.687223   -0.199219
2000-01-07  0.505503   -0.108475   -0.790826   -0.081056
2000-01-08  0.454751   -0.223420   -0.671572   -0.230215
2000-01-09  0.586390   -0.206201   -0.517619   -0.267521
2000-01-10  0.560427   -0.037597   -0.399429   -0.376886
```

.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.

Live Demo

```
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.ewm(com=0.5).mean()
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   0.865131   -0.453626   -1.137961    0.058747
2000-01-03  -0.132245   -0.807671   -0.308308   -1.491002
2000-01-04   1.084036    0.555444   -0.272119    0.480111
2000-01-05   0.425682    0.025511    0.239162   -0.153290
2000-01-06   0.245094    0.671373   -0.725025    0.163310
2000-01-07   0.288030   -0.259337   -1.183515    0.473191
2000-01-08   0.162317   -0.771884   -0.285564   -0.692001
2000-01-09   1.147156   -0.302900    0.380851   -0.607976
2000-01-10   0.600216    0.885614    0.569808   -1.110113
```

Window functions are majorly used in finding the trends within the data graphically by smoothing the curve. If there is lot of variation in the everyday data and a lot of data points are available, then taking the samples and plotting is one method and applying the window computations and plotting the graph on the results is another method. By these methods, we can smooth the curve or the trend.

Aggregations

Applying Aggregations on DataFrame

Let us create a DataFrame and apply aggregations on it.

Live Demo

```
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
r = df.rolling(window=3,min_periods=1)
print r
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   0.790670   -0.387854   -0.668132    0.267283
2000-01-03  -0.575523   -0.965025    0.060427   -2.179780
2000-01-04   1.669653    1.211759   -0.254695    1.429166
2000-01-05   0.100568   -0.236184    0.491646   -0.466081
2000-01-06   0.155172    0.992975   -1.205134    0.320958
2000-01-07   0.309468   -0.724053   -1.412446    0.627919
2000-01-08   0.099489   -1.028040    0.163206   -1.274331
2000-01-09   1.639500   -0.068443    0.714008   -0.565969
2000-01-10   0.326761    1.479841    0.664282   -1.361169

Rolling [window=3,min_periods=1,center=False,axis=0]
```

We can aggregate by passing a function to the entire DataFrame, or select a column via the standard **get item** method.

Apply Aggregation on a Whole Dataframe

Live Demo

```
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
r = df.rolling(window=3,min_periods=1)
print r.aggregate(np.sum)
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469

                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
```

Apply Aggregation on a Single Column of a Dataframe

Live Demo

```
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
r = df.rolling(window=3,min_periods=1)
print r['A'].aggregate(np.sum)
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
```

```
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
2000-01-01   1.088512
2000-01-02   1.879182
2000-01-03   1.303660
2000-01-04   1.884801
2000-01-05   1.194699
2000-01-06   1.925393
2000-01-07   0.565208
2000-01-08   0.564129
2000-01-09   2.048458
2000-01-10   2.065750
Freq: D, Name: A, dtype: float64
```

Apply Aggregation on Multiple Columns of a DataFrame

Live Demo

```
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
r = df.rolling(window=3,min_periods=1)
print r[['A','B']].aggregate(np.sum)
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
                   A           B
2000-01-01   1.088512   -0.650942
```

```
2000-01-02   1.879182   -1.038796
2000-01-03   1.303660   -2.003821
2000-01-04   1.884801   -0.141119
2000-01-05   1.194699    0.010551
2000-01-06   1.925393    1.968551
2000-01-07   0.565208    0.032738
2000-01-08   0.564129   -0.759118
2000-01-09   2.048458   -1.820537
2000-01-10   2.065750    0.383357
```

Apply Multiple Functions on a Single Column of a DataFrame

Live Demo

```
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
r = df.rolling(window=3,min_periods=1)
print r['A'].aggregate([np.sum,np.mean])
```

Its **output** is as follows −

```
                   A          B          C          D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
                   sum       mean
2000-01-01   1.088512   1.088512
2000-01-02   1.879182   0.939591
2000-01-03   1.303660   0.434553
2000-01-04   1.884801   0.628267
2000-01-05   1.194699   0.398233
2000-01-06   1.925393   0.641798
2000-01-07   0.565208   0.188403
2000-01-08   0.564129   0.188043
2000-01-09   2.048458   0.682819
2000-01-10   2.065750   0.688583
```

Apply Multiple Functions on Multiple Columns of a DataFrame

Live Demo

```
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
r = df.rolling(window=3,min_periods=1)
print r[['A','B']].aggregate([np.sum,np.mean])
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
                   A                      B
                 sum        mean         sum        mean
2000-01-01   1.088512   1.088512   -0.650942   -0.650942
2000-01-02   1.879182   0.939591   -1.038796   -0.519398
2000-01-03   1.303660   0.434553   -2.003821   -0.667940
2000-01-04   1.884801   0.628267   -0.141119   -0.047040
2000-01-05   1.194699   0.398233    0.010551    0.003517
2000-01-06   1.925393   0.641798    1.968551    0.656184
2000-01-07   0.565208   0.188403    0.032738    0.010913
2000-01-08   0.564129   0.188043   -0.759118   -0.253039
2000-01-09   2.048458   0.682819   -1.820537   -0.606846
2000-01-10   2.065750   0.688583    0.383357    0.127786
```

Apply Different Functions to Different Columns of a Dataframe

Live Demo

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(3, 4),
   index = pd.date_range('1/1/2000', periods=3),
```

```
   columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
print r.aggregate({'A' : np.sum,'B' : np.mean})
```

Its **output** is as follows −

```
                   A          B          C         D
2000-01-01  -1.575749  -1.018105   0.317797  0.545081
2000-01-02  -0.164917  -1.361068   0.258240  1.113091
2000-01-03   1.258111   1.037941  -0.047487  0.867371
                   A          B
2000-01-01  -1.575749  -1.018105
2000-01-02  -1.740666  -1.189587
2000-01-03  -0.482555  -0.447078
```

Missing Data

Missing data is always a problem in real life scenarios. Areas like machine learning and data mining face severe issues in the accuracy of their model predictions because of poor quality of data caused by missing values. In these areas, missing value treatment is a major point of focus to make their models more accurate and valid.

When and Why Is Data Missed?

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

Let us now see how we can handle missing values (say NA or NaN) using Pandas.

Live Demo

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

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
```

Its **output** is as follows −

```
        one        two      three
```

```
a   0.077988   0.476149   0.965836
b        NaN        NaN        NaN
c  -0.390208  -0.551605  -2.301950
d        NaN        NaN        NaN
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g        NaN        NaN        NaN
h   0.085100   0.532791   0.887415
```

Using reindexing, we have created a DataFrame with missing values. In the output, **NaN** means **Not a Number.**

Check for Missing Values

To make detecting missing values easier (and across different array dtypes), Pandas provides the **isnull()** and **notnull()** functions, which are also methods on Series and DataFrame objects −

Example 1

Live Demo

```
import pandas as pd
import numpy as np

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()
```

Its **output** is as follows −

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

Example 2

Live Demo

```
import pandas as pd
import numpy as np
```

```
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'].notnull()
```

Its **output** is as follows −

```
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

Example 1

Live Demo

```
import pandas as pd
import numpy as np

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()
```

Its **output** is as follows −

```
2.02357685917
```

Example 2

Live Demo

```
import pandas as pd
import numpy as np

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

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

Its **output** is as follows −

```
nan
```

Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can “fill in” NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

Replace NaN with a Scalar Value

The following program shows how you can replace "NaN" with "0".

Live Demo

```
import pandas as pd
import numpy as np

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)
```

Its **output** is as follows −

```
          one        two      three
a  -0.576991  -0.741695  0.553172
b         NaN        NaN       NaN
c   0.744328  -1.735166  1.749580

NaN replaced with '0':
          one        two      three
a  -0.576991  -0.741695  0.553172
b   0.000000   0.000000  0.000000
c   0.744328  -1.735166  1.749580
```

Here, we are filling with value zero; instead we can also fill with any other value.

Fill NA Forward and Backward

Using the concepts of filling discussed in the ReIndexing Chapter we will fill the missing values.

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

Example 1

Live Demo

```
import pandas as pd
import numpy as np

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')
```

Its **output** is as follows −

```
          one        two      three
a   0.077988   0.476149   0.965836
b   0.077988   0.476149   0.965836
c  -0.390208  -0.551605  -2.301950
d  -0.390208  -0.551605  -2.301950
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g  -0.930230  -0.670473   1.146615
h   0.085100   0.532791   0.887415
```

Example 2

Live Demo

```
import pandas as pd
import numpy as np

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='backfill')
```

Its **output** is as follows −

```
          one        two      three
a   0.077988   0.476149   0.965836
b  -0.390208  -0.551605  -2.301950
c  -0.390208  -0.551605  -2.301950
d  -2.000303  -0.788201   1.510072
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g   0.085100   0.532791   0.887415
h   0.085100   0.532791   0.887415
```

Drop Missing Values

If you want to simply exclude the missing values, then use the **dropna** function along with the **axis** argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

Example 1

Live Demo

```
import pandas as pd
import numpy as np

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.dropna()
```

Its **output** is as follows −

```
          one        two      three
a   0.077988   0.476149   0.965836
c  -0.390208  -0.551605  -2.301950
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
h   0.085100   0.532791   0.887415
```

Example 2

Live Demo

```
import pandas as pd
import numpy as np

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.dropna(axis=1)
```

Its **output** is as follows −

```
Empty DataFrame
Columns: [ ]
Index: [a, b, c, d, e, f, g, h]
```

Replace Missing (or) Generic Values

Many times, we have to replace a generic value with some specific value. We can achieve this by applying the replace method.

Replacing NA with a scalar value is equivalent behavior of the **fillna()** function.

Example 1

Live Demo

```
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})
```

Its **output** is as follows −

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

Example 2

Live Demo

```
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})
```

Its **output** is as follows −

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